Type Theories for Reactive Programming

Christian Uldal Graulund

Programming, Logic and Semantics Group
Department of Computer Science
IT University of Copenhagen

January 2021

Abstract

Functional reactive programming is the application of techniques from functional programming to the domain of reactive programming. In recent years, there has been a growing interest in *modal* functional reactive programming. Here, modal types are added to languages for functional reactive programming, with the goal of allowing the type system to enforce properties particular to reactive programming. These include *causality*, *productivity* and ruling out so-called *spaceleaks*.

The main goal of this dissertation has been to develop calculi for modal functional reactive programming, with the final aim being a fully dependent type theory for reactive programming, namely, Reactive Type Theory (RaTT). Such a system would allow a programmer to specify and verify advanced specifications for reactive systems.

The work presented here can be split into two parts: The first is concerned with Fitch-style modal calculi for synchronous (or stream based) functional reactive programming using guarded recursion and their semantics. The second describes a more domain specific modal calculus for asynchronous (or event based) functional reactive programming with widgets.

In chapter 2, the language Simply RaTT is described, which is a simply typed Fitch-style modal language for reactive programming. Both the type system and operational semantics is presented, and we prove that a well-typed term will run in the operational semantics. A special feature of the operational semantics is the aggressive garbage collection algorithm, which ensure that well-typed programs are free of unintended spaceleaks. It is further proved that the language is both causal and productive.

In chapter 3, categorical semantics for Simply RaTT is presented. The model is a Kripke-style presheaf category over a suitable category of worlds. We show that values are interpreted as coalgebras over a garbage collection modality. To interpret the $\bigcirc$ and $\Box$ modality, we use two pairs of adjoint functors. To interpret the fix point operator, we use step-indexing. Finally, we show how terms are interpreted using a reader-like monad.

In chapter 4, the language Lively RaTT is described, which is an extension of Simply RaTT. The main addition is that of temporal inductive types, which can be used to encode termination and liveness. This gives a type system that can be considered as corresponding to intuitionistic linear temporal logic, which is the natural setting for specifications of reactive properties. In particular, we show that by using a sub-modal approach, we can include temporal inductive types while retaining the ease of programming afforded by guarded recursion.

In chapter 5, the language $\lambda_{\mathsf{Widget}}$ is presented. This is a more domain specific language, aimed at programming with widgets at the abstraction level of scene graph, e.g., the DOM in a browser. This language is asynchronous and based around programming with events, and designed to have an efficient implementation strategy. The language provides a novel semantics for widgets and has a natural logical interpretation.

Resumé

Funktionel reaktive programming er anvendelsen af teknikker fra funktionle programmer til reaktiv programmering. I de seneste år har der været en voksende interesse i *modal* funktionel reaktiv programmering, hvor modale typer er tilføjet til sprog for funktionel reaktiv programmering med det formål at lade typesystemer håndhæve egenskaber reaktive egenskaber. Disse inkluderer *kausalitet*, *produktivity* og såkaldte "*spaceleaks*".

Det primære formål med denne afhandling har været at udvikle kalkyler for modal reaktive programming, med det endelige mål at være en fuld afhængig type teori for reaktiv programming, nemlig, Reaktiv Type Teori (RaTT). Sådan et system vil tillade en programmør både specificere og validere advanceret egenskaber for reaktive systemer.

Arbejdet præsenteret her er delt op i to dele: Den første omhandler Fitch-agtige modale kalklyer for synkron funktionel reaktive programmer med såkaldt "guarded" rekursion og deres semantik. Den anden beskriver en mere domænespecifik modal kalkyle for asynkron functional reaktiv programmering med "widgets".

I kapitel 2 bliver sproget Simply RaTT præsenteret, som er et Fitch-agtigt simpelt typet modalt sprog for reaktiv programming med "guarded" rekursion. Både typesystemet og den operationelle semantik beskrives, og vi beviser at vel-typet termer vil køre i den operatinelle semantik. En særlig egenskab ved den operationelle semantik er den aggresive "garbage collection" algoritme, som sikre at vel-typet programmer ikke kan indeholde utilsigtet "spaceleaks". Yderligere bevises det at sproget er både kausalt og produktivt.

I kapitel 3 præsenteres kategorisk semantik for Simply RaTT. Modellen er en "presheaf" kategori, indekseret over en passende kategori af verdener. Vi viser, hvordan værdier folkes som coalgebraer over en "garbage collection" functor. For at fortolke $\bigcirc$ og $\Box$ modaliteterne benyttes to par af adjoint funktorer. For at fortolke fikspunktoperatoren benyttes "step-indexing". Endeligt så viser vi, at termer fortolkes ved hjælp af en "reader"-agtig monade.

I kapitel 4 beskrives sproget Lively RaTT, som er en udvidelse af Simply RaTT. Den vigtigeste tilføjelse er temporale induktive typer, som kan bruges til at indkode terminering og livlighed. Dette giver et typesystem, som kan ses som værende i korrespondance til intuitionistisk linær temporal logik, som er en naturlig logik for specifikation at reaktive egenskaber. Vi viser, hvordan vi ved at bruge en "sub-modal" metode kan inkludere temporale indutive typer samtidigt med, at vi beholder den ligefremme programmeringsstil fra guarded rekursion.

I kapitel 5 beskrives sproget $\lambda_{\mathsf{Widget}}$. Dette er et mere domænespecifikt sprog som sigter mod programmering af widgets med et abstraktionsniveau, som det der bruges, når der programmeres med en scenegraf som f.eks. DOMen i en browser. Dette er et asynkront sprog, basseret på programming med "events" og designet til at have en effektiv implementering strategi. Sproget giver en klar semantik for widgets og har en naturlig logisk fortolkning.

Acknowledgements

First and foremost, I would like to thank my supervisor Rasmus Møgelberg for all the time he has spend helping and guiding me. I have spend countless hours in his office, and they have all been helpful. Secondly, I would like to thank Patrick Bahr for his role as, if only unofficially, a co-supervisor. He has helped clarify many things and his sharp attention to details has greatly improved this thesis. I would like to extend my thanks all whom I have shared an office with during my studies and the PLS group in general, for always providing a friendly and open research environment. I also would like to thank Fritz Henglein for pushing me towards doing a PhD, and for pointing out this project. Finally, none of this would have been possible without the constant support of my wife. I owe her much more than she knows.

Contents

1 Introduction **1**
1.1 Background 2
1.1.1 Data Flow Languages 2
1.1.2 Functional Reactive Programming 3
1.1.3 Modal Functional Reactive Programming 5
1.1.4 Linear Temporal Logic 5
1.1.5 Fitch-style calculi for Modal Types 6
1.1.6 Push and Pull FRP 7
1.2 Overview of Content 8
1.3 Statement of Contribution 11

2 Paper 1: Simply RaTT: A Fitch-style Modal Calculus for Reactive Programming without Spaceleaks **15**

3 A Note on Categorical Semantics for Simply RaTT **43**

4 Paper 2: Diamonds are Not Forever: Liveness in Reactive Programming with Guarded Recursion **75**

5 Paper 3: Adjoint Reactive GUI **104**

Chapter 1

Introduction

1.1 Background

As the world around us grows increasingly digital, a substantial part of our lives revolves around interactions with software of various kinds. Further, we are becoming increasingly reliant on software, and, whether we want to or not, have to put an increasing amount of trust into software. In most modern countries, almost all major institutions are highly digital, including, but not limited to, healthcare, finance, logistics, government and education. Interacting with any of these entails both direct and indirect interaction with huge amounts of software. This can be the direct interaction with online banking or the indirect interaction with modern medical equipment. If the software we interact with contains errors, we will be adversely effected, ranging from mild annoyance to mortal danger. Hence, the need for *correct* and *error free* software is greater than ever.

A certain class of software, are *reactive* systems. These are systems that have continual interaction with their *environment*. Here, an environment can mean anything that provides inputs to the system and consumes outputs from the system, e.g., a user interface or a physical environment in the case of control systems. A shared property of these systems is that they need to handle the accumulation of inputs and will often run for an indeterminate amount of time, that is, they will not terminate after some predetermined amount of time given up front. As an example, consider a graphical user interface (GUI), such as a web browser or the main interface of a smartphone. A GUI will, in general, have no natural termination point. It must be available for as long the user requires it. Additionally, it should not behave any differently after extended use, i.e., if our smartphone becomes sluggish after a day of use, we consider it a defect. Traditionally, reactive systems have been designed in an imperative fashion, using mutable shared state and networks of call-backs. While these have seen widespread use, they can be error prone, and worse, they are extremely difficult to reason about.

1.1.1 Data Flow Languages

Some of the earliest languages designed explicitly for working with reactive systems are *data flow languages*[BC85, CPHP87]. In these, a program consists of a series of *nodes*, each reading inputs and producing outputs. These inputs and outputs are usually called *signals*. Nodes are then connected using various primitives and the whole program is essentially a directed graph. One important class of data flow languages, are the *synchronous* data flow languages. These languages use the "synchronous abstraction" inspired by the analysis of digital circuits, where each node is assumed to compute its result instantaneously and each signal is assumed to transmit instantaneously. This allows the programmer to ignore the specific timing behavior of nodes and reason instead on the level of the flow of data through a program. To support this, synchronous data flow languages have a notion of an abstract, or logical, "clock". In each "tick" of the clock, all nodes receive the next input from their input signal and compute the

output. As a simple example, consider the following definition of a node

$$\begin{aligned}&\mathsf{node}\ \mathit{incNode}\ (n : \mathsf{Nat})\ \mathsf{returns}\ (m : \mathsf{Nat})\\ &\quad m = n + 1\end{aligned}$$

In each tick, the node receives a single natural number, increments it by one and transmits it.

The synchronous dataflow languages are not restricted to a specific programming paradigm, and have been designed in both imperative and declarative fashions, Esterel being an example of the former, and Lustre being an example of the latter. The use of synchronous data flow languages has been extremely successful, and has found application in the verification of controls systems from airplanes to nuclear power plants[Est19].

1.1.2 Functional Reactive Programming

Functional reactive programming(FRP)[EH97] is a more recent approach, where techniques from functional programming is applied to reactive systems. In particular, these are designed with a wish to support higher-order functions, dynamic updating of the data flow graph and ease of reasoning. In FRP, the programmer again works with *signals* and programs can now be considered *signal transducers*. As opposed to dataflow languages, FRP allows for *higher-order* transducer, which allows the programmer to, for instance, compose transducers, map over transducers, and in general, modify the data flow graph on the fly.

FRP languages can roughly be separated into two paradigms, namely *synchronous* and *asynchronous* language. In synchronous languages, the same synchronous abstraction as in synchronous data flow languages is used. That is, there is an internal clock, and the program proceeds in "ticks" of this clock. On the other hand, in asynchronous FRP, there is no such internal clock, and the program only proceeds once input from the (now asynchronous) input signal arrives. Most FRP languages are synchronous and this approach have been quite successful, but it does impose limitations on possible implementations. On the other hand, asynchronous languages allows for very efficient implementations, but has not far not received the same kind of attention.

To get a feel for the difference between the synchronous and asynchronous approach, consider the following encoding of signals as *streams*. A stream of a type is an infinite sequence of elements of that type. Given a type A, the type $\mathsf{Stream}\ A$ satisfies the type isomorphism

$$\mathsf{Stream}\ A \cong A \times \mathsf{Stream}\ A.$$

In the context of synchronous FRP, we consider the left to right unfolding as *taking time*. That is, we consider the head of the stream to be available *now* and the tail of the stream to be available in the next tick of the clock. In the context of asynchronous FRP, we will not be able to encode streams in the same way, and we will instead understand a stream as a type that produces elements with

an *indeterminate* amount of waiting in between each of these. In the following, we will be describing synchronous FRP unless otherwise stated.

Encoding signals as (synchronous) streams allows for a straight forward programming style and ease of reasoning, but does have drawbacks. In particular, the naive use of streams has some well known problems, namely *non-causality*, *non-productivity* and *spaceleaks*.

Causality is the property that each output must only depend on the previously received inputs and productivity is the property that an output will be produced in each tick. Both of these two properties should be understood in the context of the synchronous abstraction, i.e., a program can not produce an output based on inputs not yet received, and a program must produce an output in each tick. Consider the following two examples which are non-causal and non-productive, respectively:

$$\begin{aligned} &nonCausal : \mathsf{Stream}\ A \to \mathsf{Stream}\ A && nonProductive : \mathsf{Stream}\ A \\ &nonCausal = \lambda as.\mathsf{tail}\ as && nonProductive = nonProductive \end{aligned}$$

The first is non-causal since the first element of the output depends on the second element of the input. The second is non-productive, since it will never produce any outputs. Both of these are well-typed in the naive approach. Spaceleaks occur when data accumulates over time in a manner not intended by the programmer. This kind of errors are notoriously difficult to catch and may only show up after a program has run for a long time. As a somewhat contrived example, consider the following program:

$$\begin{aligned} &leaky : \mathsf{Stream\ Bool} \to \mathsf{Stream\ Nat} \to \mathsf{Stream\ Nat} \\ &leaky\ bs\ ns = \mathsf{let}\ f\ bs' = \mathsf{if}\ (\mathsf{head}\ bs') \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathsf{then}\ (\mathsf{head}\ ns) :: f\ (\mathsf{tail}\ bs') \\ &\qquad\qquad\qquad\qquad\qquad\quad \mathsf{else}\ 0 :: f\ (\mathsf{tail}\ bs') \\ &\qquad\qquad\qquad\quad \mathsf{in}\ f\ bs \end{aligned}$$

In this, the expected behavior is that at each tick, the output is 0 if the input from the first stream is false and the output is the head of the second input stream if the input from the first stream is true. The problem here is that the boolean input might *never* become true, and hence, barring an extremely eager evaluation strategy, the second input stream may be buffered *indefinitely* and thus will result in a spaceleak.

One solution is to *restrict* the direct use of streams and instead only allow programming through a set of predefined combinators. This has been the approach used in *arrowrized FRP*[NCP02], such as in the Haskell library Yampa[HCNP03]. Here, the programmer has access to a set of *signal transformers* which are assumed to be safe. While this approach is not proven free of spaceleaks, in most cases it allows for the safe use of streams, and retains higher-order nature and many of the benefits of functional programming. One problem is that is looses some of the simplicity of the original formulation of FRP.

1.1.3 Modal Functional Reactive Programming

Recently, there has been a push towards moving back to the original formulation of synchronous FRP, i.e., having direct access to signals, by using techniques from *modal type systems*[Jef12, Jef14, Jel13, Kri13, KB11, CFPP14]. In this approach, certain *modal types* are added to the type system. The most commonly used modality is the *delay* modality, usually denoted $\bigcirc$. This modality represents the passage of time on the type level, and we understand the type $\bigcirc A$ to mean "Elements of type A in the next time step". Using this, we can now work with *guarded* streams. A guarded stream $\mathsf{Stream}\ A$ satisfies the type isomorphism

$$\mathsf{Stream}\ A \cong A \times \bigcirc \mathsf{Stream}\ A.$$

This in an example of a *guarded recursive type*. These are recursive types where the recursive call is "guarded" by a modality. Having a notion the internal tick on the level of types allows for reasoning with time to some extent. For instance, it allows a type checker to reject programs that violate the linearity of time, i.e, non-causal programs. Consider again the example of a non-causal program above; with the non-guarded type of stream, this program will type check, but using guarded streams the program will be rejected. The type of tail will be $\mathsf{Stream}\ A \to \bigcirc \mathsf{Stream}\ A$, and hence, we can not use the tail of a stream to produce elements in current timestep.

A class of modal FRP languages is those that uses *guarded recursion*[Nak00]. Guarded recursion adds a fixed point operator of the type $(\triangleright A \to A) \to A$ for defining guarded recursive functions. The modality $\triangleright$ is called the *later* modality, and denotes that a type is available "later". On the face of it, the later and the delay modalities are the same, and in systems with guarded recursion, these are usually equated. This allows for a concise style of programming with guarded recursive types. Consider the following example of constructing a constant natural number stream:

$$\begin{aligned} &\mathsf{zeros} : \mathsf{Stream\ Nat} \\ &\mathsf{zeros} = \mathsf{fix}\, \mathit{zeros}'.0 :: \mathit{zeros}' \end{aligned}$$

Here zeros' has the type $\triangleright\mathsf{Stream\ Nat}$ and :: is the infix cons operator for guarded streams.

Using guarded recursion with modal FRP gives a powerful type system which ensures both causality and productivity.

1.1.4 Linear Temporal Logic

Pushing the idea of type systems for FRP further, Jeffrey [Jef12] suggested using Linear Temporal Logic (LTL) [Pnu77] as the type system for modal FRP by going through the Curry-Howard correspondence. LTL is a discrete modal temporal logic containing several modalities. In particular, it contains the *next*

modality, also denoted $\bigcirc$, the *until* modality $\mathcal{U}$ and the *always* modality $\Box$[1]. As used in LTL, the next modality denotes that a formula is true *in the next timestep* and the always modality denotes that a formula is true *in all future timesteps*. The until modality is a binary modality and denotes that a formula will be true for *some* timesteps and then another formula will be true. Importantly, the second formula in an until formula *must* become true at some point. Using the until modality, the *finally* modality $\Diamond$ can be encoded as $\Diamond\phi := \mathsf{true}\ \mathcal{U}\ \phi$. The finally modality denotes that a formula will be true *at some point*. Considered as a type system, the modalities also have clear meanings. An element of the type $\bigcirc A$ will produce an element of A in the next timestep and hence, this is precisely the expected behavior in the context of modal FRP. An element of an until type $A\ \mathcal{U}\ B$, will produce elements of A for some finite amount of time and then produce elements of B. The always modality denotes that elements of a type is available in all future timesteps. The always modality has already found use in modal FRP [Kri13] to express time-independent data and further, to rule out spaceleaks. The finally modality can be encoded in the same way using until and the type $\Diamond A$ can be understood as *terminating events* of type A. Terminating here means that an element of this type *must* return an element of A after *finitely* many steps.

1.1.5 Fitch-style calculi for Modal Types

To work with modal languages, the traditional approach has been to use some sort of split context approach [Gir93, AND92, Wad93, Bar96, Kri13] either be explicitly splitting a context into separate sections or by annotating all hypothesis with some kind of qualifier. The elimination rules are then expressed using let-bindings where variables are added to the appropriate part of the context. As an example, consider a language with the delay modality and two contexts: One for data available "now" and one for data available "later". The variable introduction rule is then restricted to only allow introduction from the now-context. The elimination and introduction rules for the next modality would be:

$$\frac{\Theta \mid \Gamma \vdash t : \bigcirc A \qquad \Theta, x : A \mid \Gamma \vdash t' : C}{\Theta \mid \Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ t' : C} \qquad \frac{\emptyset \mid \Theta \vdash t : A}{\Theta \mid \Gamma \vdash \mathsf{delay}\ t : \bigcirc A}$$

Here Θ is the "later context" and Γ is the "now context". In the elimination rule, we add a variable to the later-context, which can then be used for further typing. To introduce an element of a delay type, the contexts is *shifted*. All the now variables are removed, they are not available later as this would allow moving arbitrary data into the future and could led to spaceleaks, and all the later variables are then available now. While this approach to modal FRP has been successful, the let-binding approach has some drawbacks. In particular, if one aims to extend to dependent types, it is unclear how to handle dependencies across contexts and how to give types to the let-bindings. An alternative

[1] In *classical* LTL next and until are the only primitives. The always modality be derived from these. This is not true in intuitionistic logic and hence, always is also a primitive.

approach is the so-called *Fitch-style approach* [Gea54, Clo18]. In this approach, instead of splitting a context into parts, various *tokens* are added to a single context. The meaning of a token, and whether variable introduction is allowed across a token, depends on what modality it is associated with. In a Fitch-style presentation, the above elimination and introduction rules becomes:

$$\frac{\Gamma \vdash t : \bigcirc A}{\Gamma, \checkmark, \Gamma' \vdash \mathsf{adv}\ t : A} \qquad \frac{\Gamma, \checkmark \vdash t : A}{\Gamma \vdash \mathsf{delay}\ t : \bigcirc A}$$

Here, a single token is used. This is read as "tick" and can be understood as *a witness that time has passed*. The elimination rule then states that an element of $\bigcirc A$ can be *advanced* and used, if time has passed, i.e., there is a tick in the context. On the other hand, the introduction rule states that to produce elements of a delayed type, it should be typeable in the context with a tick, i.e., after time has passed. Note the conceptual shift between now and later in the let-binding approach and now and earlier in the Fitch-style approach. The Fitch-style approach gives a, in the authors opinion, direct and pleasant style of programming. Additionally, it is known to scale to dependent types [BGM17], and has been receiving growing attention as an alternative to the let-binding approach [CMM+18, MM18].

1.1.6 Push and Pull FRP

A stated above, most FRP languages are synchronous in the same sense as synchronous data flow languages, i.e., they have an internal notion of a tick. While this allows for clear reasoning, it forces certain restrictions on possible implementations. In particular, an implementation of synchronous FRP will need to "wake up" in each program cycle and check for new data. This approach is called *pull based*, as the data flow graph can be seen as "pulling in" data. For many classes of programs, this is not necessarily a problem. If the problem domain is itself inherently "timed", such as a control systems tied to an external clock or a game with a fixed frame rate, the pull based solution can be very efficient. On the other hand, if the problem domain is not timed, such as a GUI waiting for inputs or a server waiting for requests, the pull based solution will force the program to wake up, even though no new inputs are available. Instead, we want such programs to "sleep" and only wake up when new data arrived. This approach is called *push based*, as the data flow graph is only updated when new data is "pushed into" it.

There has been work on designing hybrid pull/push FRP languages[Ell09], but many synchronous FRP languages are by their very nature restricted to a pull based implementation. This is opposed many non-FRP systems for reactive programming, which uses implementations based on call-backs. This, for instance, is the case for many libraries for programming the Document Object Model (DOM) in the context of browsers.

1.2 Overview of Content

This thesis is a collection of three articles, one published and two accepted for publication, together with an unpublished manuscript. These can be divided into two parts.

The first part, consisting of two of the articles and the manuscript, is concerned with Fitch-style modal calculi with guarded recursion for FRP and their semantics. The overall goal of this research has been working towards a fully dependent type theory for functional reactive programming, namely Reactive Type Theory (RaTT). Having such a system, would allow a programmer to specify and verify advanced specification for reactive systems. The work as presented here is still a way off from such a system, but I believe that the work gives clear contributions towards it.

The second part, consisting of the final article, is concerned with a specific application of modal FRP, namely, the construction of GUIs using widgets and events. This work is separate from the other and presents a line of research interesting in its own right, namely asynchronous FRP and its semantics. This work is specifically concerned with designing a language allowing an efficient push based implementation.

Below, we will outline the context of each of the following chapter:

- **Simply RaTT: A Fitch-style Modal Calculus for Reactive Programming without Spaceleaks**:
  In this article, published at ICFP 2019, we present a Fitch-style modal calculus for FRP with guarded recursion. This calculus, named Simply RaTT, is loosely based on a previous calculus by Krishnaswami[Kri13]. Simply RaTT is the, to the authors knowledge, first Fitch-style calculus for FRP. The Fitch-style presentation allows for a simple and direct style of programming. We show through a series of examples that our calculus is at least as expressive as calculi with a more traditional presentation. Echoing the main result from Krishnaswami, we devise an operational semantics that ensures well-typed programs are free of spaceleaks. Further, we identity and rule out, using the Fitch-style approach, a class of *timeleaks*. Timeleaks are another kind of leaks inherent to reactive programs, and happens when a program continually recomputes results.

  The main contributions of the article is a Fitch-style type system, two heap based operational semantics and a novel Kripke-style logical relation. Overall, the Fitch-style presentation is a significant simplification over more earlier approaches.

- **A Note On Categorical Semantics for Simply RaTT**:
  In this unpublished manuscript, I give categorical semantics for Simply RaTT. In particular, I construct a Kripke style presheaf model with worlds corresponding to the worlds used in the logical relation construction in the Simply RaTT work.

Having a categorical model for the full language, one would be able to consider the internal logic of the model. Such a *reactive* logic would be useful for creating specifications for reactive programs with the same guarantees about spaceleaks and would aid in the future design of a dependent type theory.

In the model, I give a series of abstract constructions needed for interpreting Simply RaTT. I define a *garbage collection* functor GC, which is additionally an idempotent comonad, and show that values are interpreted as coalgebras over this comonad. I show that it follows from this structure that for all values $A \cong \mathsf{GC}(A)$.

To give the interpretation of the $\bigcirc$ modality, I use a pair of adjoint "shifting" functors, $\Downarrow \dashv \Uparrow$, which work on contexts and values, and gives an abstract version of a timestep. These shifting functors have non-trivial interaction with the garbage collection modality, and getting the correct interpretation relies on this interaction.

To give the interpretation of the $\Box$ modality, I again use a pair of adjoint functor, $\sqcup \dashv \sqcap$, which works on contexts and general terms, and give an abstract version of "time-independent" data.

Further, I show that terms are interpreted using a commutative "reader-like" monad over a suitable store object. This monad again interacts with both the above functors and garbage collection in non-trivial ways.

To give the interpretation of the fix point operator, I use an version of step-indexing where termination is ensured by definition of the $\Uparrow$ functor.

- **Diamonds are not forever: Liveness in Reactive Programming with Guarded Recursion**:
  In this article, accepted for publication at POPL 21, we consider how to use linear temporal logic (LTL) as a type system for FRP in the presence of guarded recursion. It is well known that in systems with guarded recursion least and greatest fixpoints coincide [BMSS11], and in fact, they can be made to behave more like the latter, as shown by Atkey and McBride [AM13]. If we naively consider the delay modality $\bigcirc$ used in modal FRP to be equal to the later modality $\triangleright$ used for guarded recursion, we loose the ability to encode the correct inductive behavior of LTL and hence, we can not give the correct meaning to the modalities.

  Consider the until modality $A \,\mathcal{U}\, B$, which should satisfy the type isomorphism:

  $$A \,\mathcal{U}\, B \cong B + (A \times \bigcirc(A \,\mathcal{U}\, B))$$

  In a language with guarded recursion and $\triangleright = \bigcirc$, the following program is well typed

  $$\lambda(a : A).\mathsf{fix}(u : \bigcirc(A \,\mathcal{U}\, B)).\mathsf{inr}(\langle a, u\rangle)$$

but this allows us to produce elements of $A \,\mathcal{U}\, B$ that only contain elements of A and hence *does not* model the correct behavior.

Instead, we propose to consider the delay modality a *submodality* of the later modality, and restricting the use of guarded recursion to the latter. This way, we retain the ease of programming with guarded recursion, while getting the correct inductive behavior of until types. In particular, we show that we have an embedding map in one direction, $\bigcirc A \to \triangleright A$, but, in general, not the other. We build upon the language of Simply RaTT to get the language Lively RaTT which contains both $\bigcirc A$, $\triangleright A$ and $A \,\mathcal{U}\, B$. We similarly extend the operational semantics and prove that terms of type $A \,\mathcal{U}\, B$ *will in fact terminate*.

We use a similar logical relation technique as for Simply RaTT, but extend the worlds to include an additional step-index, used for inductive types.

- **Adjoint Reactive GUI**:
  In this article, accepted for publication at FoSSaCS 2021, we devise an asynchronous FRP language, called $\lambda_{\mathsf{Widget}}$, aimed at defining GUIs using widgets and events. The goal of this research is creating FRP languages that allow for the same kind of straight forward programming and reasoning as for synchronous FRP languages, but also allows an efficient push-based implementation using call-backs.

  One of the major differences between the previous synchronous languages and $\lambda_{\mathsf{Widget}}$ is that a program *does not* have access to an internal "tick". Instead, all future data is encoded using *events*. To work with events, the programmer use the select construction, which can be seen as a kind of case-statement, which must respect linearity of time. Having multiple events, the programmer must consider all possibilities for which event arrives first, and must handle them accordingly.

  We give a categorical semantics of our language in terms of a linear-non-linear hyperdoctrine. While the model is technically advanced, the approach is entirely standard, and extends upon well-known interpretations of both temporal logic and linear logic. In particular, we show the semantics of widgets being especially simple. The semantics is given by a "logbook" which simply records all commands applied to the object. At each timestep, the state of the widget object can be given by "replaying" the history of the object.

  Further, we show how the type system of our language has a straightforward logical interpretation, namely, as a linear, linear temporal logic satisfying the S4.3 axioms.

  Finally, the concrete language is presented through a series of programming examples, showing the straightforward nature of the language.

1.3 Statement of Contribution

- P. Bahr, C. Graulund and R.E. Møgelberg: *Simply RaTT: A Fitch-style Modal Calculus for Reactive Programming without Spaceleaks*. Published at ICFP 2019:

  I participated in the design of the type system and operational semantics. Further, I contributed most of the proof for the fundamental property. Finally, I collaborated in writing the article and provided most of the example programs.

- C. Graulund: *A Note on Categorical Semantics of Simply RaTT*. Unpublished manuscript:

  Fully authored by myself.

- P. Bahr, C. Graulund and R.E. Møgelberg: *Diamonds are not forever: Liveness in reactive programming with guarded recursion*. Accepted for publication at POPL 2021:

  I participated in the design of the type system, logical relation and operational semantics. Further, I contributed most of the proof of the fundamental property. Finally, I collaborated in writing the article with responsibility for most of the meta-theory section and example programs.

- C. Graulund, D. Szamozvancev and N. Krishnaswami: *Adjoint Reactive GUI*. Accepted for publication at FoSSaCS 2021:

  I participated in the design of the type system and contributed most of the syntactic lemmas and part of the categorical model. Further, I collaborated in writing the article with responsibility for most of the sections.

Bibliography

[AM13] Robert Atkey and Conor McBride. Productive coprogramming with guarded recursion. *SIGPLAN Not.*, 48(9):197–208, September 2013.

[AND92] JEAN-MARC ANDREOLI. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation*, 2(3):297–347, 06 1992.

[Bar96] Andrew Barber. Dual intuitionistic linear logic. Technical report, University of Edinburgh, Edinburgh, UK, 1996.

[BC85] Gérard Berry and Laurent Cosserat. The esterel synchronous programming language and its mathematical semantics. In Stephen D. Brookes, Andrew William Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, pages 389–448, Berlin, Heidelberg, DE, 1985. Springer Berlin Heidelberg.

[BGM17] Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. The clocks are ticking: No more delays! In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12, Washington, DC, USA, 2017. IEEE Computer Society.

[BMSS11] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. In *In Proc. of LICS*, pages 55–64, Washington, DC, USA, 2011. IEEE Computer Society.

[CFPP14] Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. Fair Reactive Programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 361–372, San Diego, California, USA, 2014. ACM.

[Clo18] Ranald Clouston. Fitch-style modal lambda calculi. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 258–275, Cham, 2018. Springer International Publishing.

[CMM+18] Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. Modal dependent type theory and dependent right adjoints. *CoRR*, abs/1804.05236:1–21, 2018.

[CPHP87] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: A declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 178–188, New York, NY, USA, 1987. ACM.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM.

[Ell09] Conal M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, Haskell '09, page 25–36, New York, NY, USA, 2009. Association for Computing Machinery.

[Est19] Esterel Technologies SA. Success stories. `http://www.esterel-technologies.com/success-stories/`, 2019.

[Gea54] P. T. Geach. Fitch, f. b. -symbolic logic, an introduction. *Mind*, 63(n/a):274, 1954.

[Gir93] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59(3):201 – 217, 1993.

[HCNP03] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. *Arrows, Robots, and Functional Reactive Programming*, pages 159–187. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[Jef12] Alan Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In Koen Claessen and Nikhil Swamy, editors, *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*, pages 49–60, Philadelphia, PA, USA, 2012. ACM.

[Jef14] Alan Jeffrey. Functional reactive types. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 54:1–54:9, New York, NY, USA, 2014. ACM.

[Jel13] Wolfgang Jeltsch. Temporal logic with "until", functional reactive programming with processes, and concrete process categories. In *Proceedings of the 7th Workshop on Programming Languages Meets*

Program Verification, PLPV '13, pages 69–78, New York, NY, USA, 2013. ACM.

[KB11] N. R. Krishnaswami and N. Benton. Ultrametric semantics of reactive programs. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 257–266, Washington, DC, USA, June 2011. IEEE Computer Society.

[Kri13] Neelakantan R. Krishnaswami. Higher-order Functional Reactive Programming Without Spacetime Leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 221–232, Boston, Massachusetts, USA, 2013. ACM.

[MM18] Bassel Mannaa and Rasmus Ejlers Møgelberg. The clocks they are adjunctions denotational semantics for clocked type theory. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*, pages 23:1–23:17, New York, NY, USA, 2018. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[Nak00] Hiroshi Nakano. A modality for recursion. *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pages 255–266, 2000.

[NCP02] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, page 51–64, New York, NY, USA, 2002. Association for Computing Machinery.

[Pnu77] Amir Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[Wad93] Philip Wadler. A taste of linear logic. In Andrzej M. Borzyszkowski and Stefan Sokołowski, editors, *Mathematical Foundations of Computer Science 1993*, pages 185–210, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

Chapter 2

Paper 1: Simply RaTT: A Fitch-style Modal Calculus for Reactive Programming without Spaceleaks

Simply RaTT: A Fitch-Style Modal Calculus for Reactive Programming without Space Leaks

PATRICK BAHR, IT University of Copenhagen, Denmark
CHRISTIAN ULDAL GRAULUND, IT University of Copenhagen, Denmark
RASMUS EJLERS MØGELBERG, IT University of Copenhagen, Denmark

Functional reactive programming (FRP) is a paradigm for programming with signals and events, allowing the user to describe reactive programs on a high level of abstraction. For this to make sense, an FRP language must ensure that all programs are causal, and can be implemented without introducing space leaks and time leaks. To this end, some FRP languages do not give direct access to signals, but just to signal functions.

Recently, modal types have been suggested as an alternative approach to ensuring causality in FRP languages in the synchronous case, giving direct access to the signal and event abstractions. This paper presents *Simply RaTT*, a new modal calculus for reactive programming. Unlike prior calculi, Simply RaTT uses a Fitch-style approach to modal types, which simplifies the type system and makes programs more concise. Echoing a previous result by Krishnaswami for a different language, we devise an operational semantics that safely executes Simply RaTT programs without space leaks.

We also identify a source of time leaks present in other modal FRP languages: The unfolding of fixed points in delayed computations. The Fitch-style presentation allows an easy way to rules out these leaks, which appears not to be possible in the more traditional dual context approach.

CCS Concepts: • **Software and its engineering** → **Functional languages**; **Data flow languages**; *Recursion*; • **Theory of computation** → *Operational semantics.*

Additional Key Words and Phrases: Functional reactive programming, Modal types, Synchronous data flow languages, Type systems, Garbage collection

ACM Reference Format:
Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2019. Simply RaTT: A Fitch-Style Modal Calculus for Reactive Programming without Space Leaks. *Proc. ACM Program. Lang.* 3, ICFP, Article 109 (August 2019), 27 pages. https://doi.org/10.1145/3341713

1 INTRODUCTION

Reactive programs are programs that engage in an ongoing dialogue with their environment, taking inputs and producing outputs, typically dependent on an internal state. Examples include GUIs, servers, and control software for components in cars, aircraft, and robots. These are traditionally implemented in imperative programming languages using often complex webs of components communicating through callbacks and shared state. As a consequence, reactive programming in imperative languages is error-prone and program behaviour difficult to reason about. This is unfortunate since many of the most safety-critical programs in use today are reactive.

Authors' addresses: Patrick Bahr, IT University of Copenhagen, Denmark, paba@itu.dk; Christian Uldal Graulund, IT University of Copenhagen, Denmark, cgra@itu.dk; Rasmus Ejlers Møgelberg, IT University of Copenhagen, Denmark, mogel@itu.dk.

2475-1421/2019/8-ART109
https://doi.org/10.1145/3341713

The idea of Functional Reactive Programming (FRP) [Elliott and Hudak 1997] is to bring reactive programming into the functional paradigm by providing the programmer with abstractions for describing the dataflow between components in a simple and direct way. At the same time, this should give the usual benefits of functional programming: Modular programming using higher-order functions, and simple equational reasoning. The abstractions provided by the early FRP languages were *signals* and *events*: A signal of type A is a time-varying value of type A, and an event of type A is a value of type A appearing at some point in time. The notion of time is abstract, but can, depending on the application, be thought of as either discrete or continuous.

For such high-level abstractions to make sense, the language designer must ensure that all programs can be executed in an efficient way. A first problem is ensuring *causality*, i.e., the property that the value of output signals at a given time only depends on the values read from input signals before or at that time. For example, implementing signals in the discrete time case simply as streams will break this abstraction, as there are many non-causal functions from streams to streams. Another issue is *time leaks*, i.e., the problem of programs exhibiting gradually slower response time, typically due to intermediate values being recomputed whenever output is needed. The related notion of *space leaks* is the problem of programs holding on to memory while continually allocating more until they eventually run out of memory.

A good language for FRP should only allow programmers to write causal functions. On the other hand, in expressive programming languages some of the responsibility for avoiding the problems of space and time leaks must be left to the programmer. For example, if the language has linked lists, a programmer could write a function that stores all input in a list, leading to a space leak. We will refer to this as an *explicit* space leak, since it can be detected from the code. A good FRP language should avoid *implicit* space and time leaks, i.e., leaks that are caused by the language implementation, and so are out of the programmers control.

Due to these concerns, newer libraries and languages for FRP do not give the programmer direct access to signal and event types. For example, Arrowised FRP [Nilsson et al. 2002] has a primitive notion of signal functions and provides combinators for combining these to construct dataflow networks statically, along with switching operators for dynamically changing these networks. This approach sacrifices some of the simplicity and flexibility of the original suggestions for FRP, and the switching combinators have an ad hoc flavour. Moreover, to the best of our knowledge, no strong guarantees concerning space or time leaks have been proved in this setting.

1.1 Modal FRP Calculi

Recently, a number of authors ([Jeffrey 2012, 2014; Jeltsch 2013; Krishnaswami 2013; Krishnaswami and Benton 2011]) have suggested using modal types for functional reactive programming. These all work in the synchronous case of time being given by a global clock. With this assumption, the resulting languages can be thought of as extensions of synchronous dataflow languages such as Lustre [Caspi et al. 1987], and Lucid Synchrone [Pouzet 2006] with higher-order functions and operations for dynamically changing the dataflow network. This restricted setting covers many applications of FRP, and in this paper we shall restrict ourselves to that as well. Since continuous time can be simulated by discrete time (see Section 7), we will further restrict ourselves to discrete time.

Under the assumption of a global discrete clock a signal is simply a stream. Causality is ensured by using a modal type operator $\bigcirc$ to encode the notion of a time step in types: A value of type $\bigcirc A$ is a computation returning a value of type A in the next time step. Using $\bigcirc$, one can describe the streams corresponding to signals as a type satisfying the type isomorphism $\mathrm{Str}(A) \cong A \times \bigcirc\mathrm{Str}(A)$, capturing the fact that the tail of the stream is only available in the next time step. Streams and

programs processing streams can be defined recursively using the guarded fixed point combinator of Nakano [2000] taking input of type $\bigcirc A \to A$ and producing elements of type A as output.

The most advanced programming language of this kind, in terms of operational semantics with run-time guarantees, is that of Krishnaswami [2013]. This language extends the simply typed lambda calculus with two modal type operators: The $\bigcirc$ mentioned above, as well as one for classifying stable, i.e., time-invariant data. Unlike the arrowised approach to FRP, Krishnaswami's calculus gives direct access to streams as a data type which can even be nested to give streams of streams. Other important data types, such as events can be encoded using *guarded recursive types*, a concept also stemming from Nakano [2000].

Krishnaswami's calculus has an operational semantics for evaluating terms in each step of the global clock, and this can be extended to a step-by-step evaluation of streams. The language is total in the sense that each step evaluates to a value in finite time (a property often referred to as *productivity*). The operational semantics evaluates by storing delayed computations on a heap, and Krishnaswami shows that all heap data can be safely garbage collected after each evaluation step, effectively guaranteeing the absence of (implicit) space leaks.

1.2 Fitch-style Modal Calculi

Like most modal calculi, Krishnaswami's calculus uses let-expressions to program with modalities. This affects the programming style: Many programs consist of a long series of unpacking statements, essentially giving access to the values produced by delayed computations in the next time step, followed by relatively short expressions manipulating these. While this can be to a large extent be dealt with using syntactic sugar, it has a more fundamental problem, which is harder to deal with: It complicates equational reasoning about programs. This is an important issue, since simple equational reasoning is supposed to be one of the benefits of functional programming. Our long-term goal is to design a dependent type theory for reactive programming in which programs have operational guarantees like the ones proved by Krishnaswami, and where program specifications can be expressed using dependent types. Introducing let-expressions in terms will lead to let-expressions also in the types, which is a severe complication of the type theory.

Fitch-style modal calculi [Clouston 2018; Fitch 1952] are an alternative approach to modal types not using let-expressions. Instead, elements of modal types are constructed by abstracting tokens from the context, and modal operators are likewise eliminated by placing tokens in the context. Recent research in guarded dependent type theory [Bahr et al. 2017; Clouston et al. 2018] has shown the benefit of this approach also for dependent types. Guarded dependent type theory is an extension of Martin-Löf type theory [Martin-Löf and Sambin 1984] with a delay modality reminiscent of the $\bigcirc$ used in modal FRP together with Nakano's fixed point combinator also mentioned above. In this setting, the token used in the Fitch-style approach is thought of as a 'tick' – evidence that time has passed – which can be used to open a delayed computation. Using ticks, one can prove properties of guarded recursive programs in a compellingly simple way.

1.3 Simply RaTT

In this paper, we present Simply Typed Reactive Type Theory (Simply RaTT) a simply typed calculus for reactive programming based on the Fitch-style approach to modal types. This is a first step towards our goal of a dependently typed theory for reactive programming (RaTT), but already the simply typed version offers several benefits over existing approaches. Compared to Krishnaswami's calculus, Simply RaTT has a significantly simpler type system. The Fitch-style approach eliminates the need for the qualifiers 'now', 'later' and 'stable' used in Krishnaswami's calculus on variables and term judgements. Similar (but not quite the same) qualifiers can be derived from the position of variables relative to tokens in contexts in Simply RaTT. Moreover, we eliminate the need for

allocation tokens, a technical tool used by Krishnaswami to control heap allocation. This, together with the Fitch-style typing rules makes programs shorter and (we believe) more readable than in Krishnaswami's calculus.

Compared to the standard approach to modal types, the Fitch-style used here is based on a shift in time-dependence. Whereas terms in Krishnaswami's language can look into the future (since now-terms can depend on later-variables), terms in Simply RaTT can only look into the past (since later-expressions can depend on now-variables). This explains how let-expressions are eliminated: There is no need to refer to the values produced in the future by delayed computations. Instead, Simply RaTT allows delayed computations from the past to be run in the present.

We prove a garbage collection result similar to that proved by Krishnaswami, and show how this can be used to construct a safe evaluation strategy for stream transducers written in our language. Input to stream transducers are treated as delayed computations, and therefore stored in a heap and garbage collected in the next time step.

We also identify and eliminate a source of time leaks present in previous approaches. This is best illustrated by the following two implementations of the stream of natural numbers written in Haskell-notation:

$$leakyNats = 0 :: map\ (+1)\ leakyNats \qquad nats = from\ 0 \ \textbf{where}\ from\ n = n :: from\ (n+1)$$

On most machines (some compilers may use clever techniques to detect this problem), the evaluation of the nth element of *leakyNats* will not use the previously computed values, but instead compute it using n successive applications of *suc*, resulting in a time leak. This is indeed what happens on Krishnaswami's machine and also the machine of this paper. Contrary to that, the *nats* example uses an internal state declared explicitly in the type of *from* to maintain a constant evaluation time for each step. In this paper we identify the source of the time leak to be the ability to unfold fixed points in delayed computations, and use this to eliminate examples such as *leakyNats* in typing. The ability to control when unfolding of fixed points are allowed relies crucially on the Fitch-style presentation, and it is very unclear whether a similar restriction can be added to the traditional dual context presentation.

The calculus is illustrated through examples showing how to implement a small FRP library as well as how to simulate the most basic constructions of Lustre in Simply RaTT. Examples are also used to illustrate our abstract machine for evaluating streams and stream transducers.

1.4 Overview of Paper

The paper is organised as follows: Section 2 gives an overview of the language introducing the main concepts and their intuitions through examples. Section 3 defines the operational semantics, including the evaluation of stream transducers and states the garbage collection results for these. Section 4 shows how to implement a small library for reactive programming in Simply RaTT and Section 5 shows how to encode the most basic constructions of the synchronous dataflow language Lustre in Simply RaTT. Section 6 sketches the proof of our garbage collection result. The metatheory presented in Section 6 has been fully formalised in the accompanying Coq proofs. Finally, Section 7 describes related work and Section 8 concludes and describes future work.

2 SIMPLY RATT

This section gives an overview of the Simply RaTT language. The complete formal description of the syntax of the language, and in particular the typing rules, can be found in Figure 2, Figure 3, and Figure 4.

$$\Gamma \vdash t : A \qquad \Gamma, \sharp, \Gamma_N \vdash t : A \qquad \Gamma, \sharp, \Gamma_N, \checkmark, \Gamma_L \vdash t : A$$

(a) Initial judgement (b) Now judgement (c) Later judgement

Fig. 1. The different type judgement forms. In these, the contexts Γ, Γ_N and Γ_L are assumed to be token-free and contain variables referred to as initial variables, now-variables and, later-variables.

The type system of Simply RaTT extends that of the simply typed lambda calculus with two modal type operators: $\bigcirc$ for classifying delayed computations, and $\Box$ for classifying stable computations, i.e., computations that can be performed safely at any time in the future. We start by describing the constructions for $\bigcirc$.

Data of type $\bigcirc A$ are *computations* that produce data of type A in the next time step. To perform such a computation we must wait a time step, as represented in typing judgements by the addition of a $\checkmark$ (pronounced 'tick') in the context. More precisely, the typing rule for eliminating $\bigcirc$ states that if $\Gamma \vdash t : \bigcirc A$ then $\Gamma, \checkmark, \Gamma' \vdash \mathsf{adv}(t) : A$. The $\checkmark$ in the context of $\mathsf{adv}(t)$ should be thought of as separating variables in time: Those in Γ are available one time step before those in Γ'. Since there can be at most one $\checkmark$ in a context, we will refer to these times as 'now' and 'later'. The typing assumption on t states that it has type $\bigcirc A$ now, and the conclusion states that $\mathsf{adv}(t)$ has type A later. The constructor for $\bigcirc A$ states that if $\Gamma, \checkmark \vdash t : A$, i.e., if t has type A later, but depends only on variables available now, then it can be turned into a *thunk* $\mathsf{delay}(t)$ of type $\bigcirc A$ now.

Note that terms in 'later' judgements can refer to variables available now as well as later, but 'now' judgements can only refer to variables available now. This separates the Fitch-style approach of Simply RaTT from the traditional dual context approach to calculi with modalities, such as Krishnaswami's [2013] modal calculus for reactive programming. The latter also has a distinction between 'later' and 'now', but the time dependencies work the opposite way: A later-judgement can only depend on later-variables, whereas a now-judgement can depend on both now- and later-variables.

Data of type $\Box A$ are time invariant *computations* that produce data of type A. That is, these computations can be executed safely at any time in the future. To allow time invariant computations to depend on initial data, that is, data available before the reactive program starts executing, contexts may contain a $\sharp$ separating the context into *initial variables* (those to the left of $\sharp$) and *temporal variables* to the right of $\sharp$. There can be at most one $\sharp$ in a context, and $\Gamma, \checkmark$ is only well-formed if there is a $\sharp$ in Γ. Thus $\checkmark$ separates the temporal variables into now and later. We refer collectively to $\checkmark$ and $\sharp$ as *tokens*. Judgements in a token-free context is referred to as an initial judgement. The three kinds of judgements are summarised in Figure 1.

If $\Gamma, \sharp \vdash t : A$ then t does not depend on any temporal data, and can thus be thunked to a time invariant computation $\Gamma \vdash \mathsf{box}(t) : \Box A$ to be run at a later time. The typing rule for eliminating $\Box$ states that if $\Gamma \vdash t : \Box A$ and Γ' is token-free, then $\Gamma, \sharp, \Gamma' \vdash \mathsf{unbox}(t) : A$. The restriction on Γ' means that we can only run the time invariant computation t now, not later. This may seem to contradict the intuition for $\Box A$ given above, but is needed to rule out certain time leaks as we shall see below. Time invariant computations can still be run at arbitrary times in the future through the use of fixed points.

Both these modal type operators have restricted forms of applicative actions. In the case of $\bigcirc$, if $\Gamma \vdash t : \bigcirc(A \to B)$ and $\Gamma \vdash u : \bigcirc A$ then $\Gamma \vdash t \circledast u : \bigcirc B$ is defined as

$$t \circledast u = \mathsf{delay}(\mathsf{adv}(t)(\mathsf{adv}(u))).$$

Note that this is only well-typed if Γ contains $\sharp$ but not $\checkmark$, since the subterm $\mathsf{adv}(f)(\mathsf{adv}(x))$ must be typed in context $\Gamma, \checkmark$, and by the restrictions mentioned above, this is only a well-formed context if $\sharp$ is the only token in Γ. Similarly, if $\Gamma \vdash t : \Box(A \to B)$ and $\Gamma \vdash u : \Box A$ then $\Gamma \vdash t \circledast u : \Box B$ is defined as $\mathsf{box}(\mathsf{unbox}(t)(\mathsf{unbox}(u)))$. As above, this is only well-typed if Γ is token-free. Note that neither $\Box$ nor $\bigcirc$ are applicative functors in the sense of McBride and Paterson [2008], since there are generally no maps $A \to \Box A$, nor $A \to \bigcirc A$. The former would force computations to be stable, and the latter would push data into the future, which is generally unsafe as it can lead to space leaks. This restriction is enforced in the type theory in the variable introduction rule, which does not allow variables to be introduced over tokens. As a consequence, weakening of typing judgements with tokens is not admissible. An exception to this exists for the *stable* types, as we shall see below.

2.1 Fixed Points

Reactive programs can be defined recursively using a fixed point combinator. To ensure productivity and causality, the recursion variable must be a delayed computation. Precisely, the rule for fixed points state that if $\Gamma, \sharp, x : \bigcirc A \vdash t : A$ then $\Gamma \vdash \mathsf{fix}\, x.t : \Box A$. These guarded recursive fixed points can be used to program with guarded recursive types such as guarded recursive streams $\mathsf{Str}(A)$ satisfying the type isomorphism $\mathsf{Str}(A) \cong A \times \bigcirc\mathsf{Str}(A)$. Terms of this type compute to an element in A (the head) now, and a delayed computation of a tail. We will use :: as infix notation for the right to left direction of the isomorphism, i.e., $t :: u$ is a shorthand for into $\langle t, u\rangle$. Given $t : A$ and $u : \bigcirc\mathsf{Str}(A)$, we thus have $t :: u : \mathsf{Str}(A)$.

As a simple example of a recursive definition, the stream of all zeros can be defined as

$$zeros = \mathsf{fix}\ x\,.\,0 :: x : \Box\ (\mathsf{Str}\ (\mathsf{Nat}))$$

Note that fixed points are time invariant in the sense of having a type of the form $\Box A$. This is because they essentially need to call themselves in the future. For this reason, their definition cannot depend on temporal data, as can be seen from the typing rule, since x must be the only temporal variable in t.

As a second example of a recursively defined function, we define a map function for guarded streams. This should take a function $A \to B$ as input and a stream of type $\mathsf{Str}(A)$ and produce a stream of type $\mathsf{Str}(B)$. Since the input function will be called repeatedly at all future time steps it needs to be time-invariant, and can be defined as:

$$map : \Box\ (A \to B) \to \Box\ (\mathsf{Str}\ A \to \mathsf{Str}\ B)$$
$$map = \lambda f\,.\,\mathsf{fix}\ x\,.\,\lambda as\,.\,\mathsf{unbox}\ f\ (\mathsf{head}\ as) :: x \circledast \mathsf{tail}\ as$$

where head and tail compute the head and the tail of a stream, respectively.

For readability we introduce the following syntax for defining fixed points such as *map*:

$$map\ f\ \sharp\ (a :: as) = \mathsf{unbox}\ f\ \ a :: map\ f \circledast as$$

This should be read as defining the term to the left of $\sharp$ as a fixed point and in particular it allows us to write pattern matching in a simple way. When type checking the right-hand side of this definition, $map\ f$ should be given type $\bigcirc(\mathsf{Str}(A) \to \mathsf{Str}(B))$ because it represents the recursion variable. Any such definition can be translated syntactically to our core language in a straightforward manner: Pattern matching is translated to the corresponding elimination forms (π_i, case, out) and the recursion syntax with $\sharp$ is translated to fix.

The type of guarded streams defined above is just one example of a guarded recursive type. Simply RaTT includes a construction for general recursive types $\mu\alpha.A$ satisfying type isomorphisms of the form $\mu\alpha.A \cong A[\bigcirc(\mu\alpha.A)/\alpha]$. In these α can appear everywhere in A, including non-strictly positive and even negative positions. Another example of a guarded recursive type is that of events

defined as $\mathsf{Ev}(A) = \mu\alpha.A + \alpha$, and thus satisfying $\mathsf{Ev}(A) \cong A + \bigcirc\mathsf{Ev}(A)$. Streams and events form the building blocks of functional reactive programming. Similarly to streams, one can define a map function for events using fixed points as follows

$$
\begin{aligned}
&\mathit{map} : \Box\,(A \to B) \to \Box\,(\mathsf{Ev}\ A \to \mathsf{Ev}\ B)\\
&\mathit{map}\ f\ \sharp\,(\mathsf{wait}\ \mathit{eva}) = \mathsf{wait}\ (\mathit{map}\ f \circledast \mathit{eva})\\
&\mathit{map}\ f\ \sharp\,(\mathsf{val}\ a) \quad\ = \mathsf{val}\ (f\ a)
\end{aligned}
$$

where we write $\mathsf{val}\,t$ and $\mathsf{wait}\,t$ instead of $\mathsf{into}\,(\mathsf{in}_1\,t)$ and $\mathsf{into}\,(\mathsf{in}_2\,t)$, respectively.

2.2 Stable Types

Next we show how to define the stream of natural numbers using a helper function mapping a natural number n to the stream $(n, n+1, n+2, \dots)$. A first attempt at defining *from* could look as follows:

$$
\begin{aligned}
&\mathit{from} : \Box\,(\mathsf{Nat} \to \mathsf{Str}\,(\mathsf{Nat}))\\
&\mathit{from}\,\sharp\,n = n :: \mathit{from} \circledast \mathsf{delay}\ (n+1)
\end{aligned}
$$

is not well typed, because to type $\mathsf{delay}(n+1)$ the term $n+1$ must have type Nat *later*, but n is a *now*-variable. The number n therefore must be kept for the next time step, an operation that generally is unsafe, because general values can have references to temporal data. For example, a value of type $\bigcirc\mathsf{Str}(A)$ in our machine is a reference to the tail of a stream, which could be an input stream. Allowing such values to be kept for the next step can lead the machine to store input data indefinitely, causing space leaks. Similarly, values of function types can contain references to time dependent data in closures and should therefore not be kept. On the other hand, a value of type natural numbers cannot contain such references and so can safely be kept for the next time step. We say that Nat is a *stable* type, and a grammar for these stable types is given in Figure 3. Data of stable type can be kept one time step using the construction progress which allows a now-judgement of the form $\Gamma \vdash t : A$ to be transformed to a later judgement of the form $\Gamma, \checkmark, \Gamma' \vdash \mathsf{progress}\,t : A$ if Γ contains a $\sharp$ and no $\checkmark$ and if A is stable. In our operational semantics, $\mathsf{progress}\,t$ evaluates by evaluating t to a value now pushing the result to the future. Postponing the evaluation of t would be unsafe, since terms of stable types, unlike values of stable types, can refer to temporal data. Similarly, promote can be used to make stable initial data available in temporal judgements.

We introduce the constructions $\odot$, defined as $t \odot u = \mathsf{delay}(\mathsf{adv}(t)(\mathsf{progress}\,u))$, and $\boxdot$, defined as $t \boxdot u = \mathsf{box}(\mathsf{unbox}(t)(\mathsf{promote}\,u))$, with derived typing rules

$$
\frac{\Gamma \vdash t : \bigcirc(A \to B) \qquad \Gamma \vdash u : A \qquad \Gamma, \checkmark \vdash \qquad A\ \mathsf{stable}}{\Gamma \vdash t \odot u : \bigcirc B}
$$

$$
\frac{\Gamma \vdash t : \Box(A \to B) \qquad \Gamma \vdash u : A \qquad \Gamma, \sharp \vdash \qquad A\ \mathsf{stable}}{\Gamma \vdash t \boxdot u : \Box B}
$$

Using this, *from* and *nats* can be defined as follows

$$
\begin{aligned}
&\mathit{from} : \Box\,(\mathsf{Nat} \to \mathsf{Str}\ \mathsf{Nat}) && \mathit{nats} = \Box\,(\mathsf{Str}\ \mathsf{Nat})\\
&\mathit{from}\,\sharp\,n = n :: \mathit{from} \odot (n+1) && \mathit{nats} = \mathit{from} \boxdot 0
\end{aligned}
$$

Many programming languages would also allow *nats* to be defined directly as fixed point as $\mathit{leakyNats} = 0 :: \mathit{map}\ (+1)\ \mathit{leakyNats}$. In Simply RaTT, however, such a definition would not be well typed, because the term $\mathit{map}(\mathsf{box}(+1))$ of type $\Box(\mathsf{Str}(\mathsf{Nat}) \to \mathsf{Str}(\mathsf{Nat}))$ would have to be unboxed

in a context with a ✓ in order to type a term like

$$\mathit{leakyNats}\,\sharp = 0 :: \mathsf{delay}(\mathsf{unbox}(\mathit{map}(\mathsf{box}(+1)))) \circledast \mathit{leakyNats}$$

and this is not allowed according to the typing rule for unbox. We believe such a definition should be ruled out because it leads to time leaks as explained in the introduction. This indeed happens on the machined described in Section 3 as well as the machine of Krishnaswami [2013].

The time leak in the *leakyNats* example above happens because the fixed point definition of *map* is unfolded in a delayed term, allowing the term to be evaluated to grow for each iteration. In the *nats* example, on the other hand, the recursive definition uses a state, namely the input to *from*, to avoid repeating computations. Moreover, this state usage is essentially declared in the type of *from*.

For similar reasons, the *scary_const* example of Krishnaswami [2013] in which all data from an input stream is kept indefinitely by explicitly storing it in a stream of streams cannot be typed in Simply RaTT. An implementation of *scary_const* in Simply RaTT would require an explicit state that stores all previous elements from the input stream. That could be achieved by extending the language to include a list type List A, and defining that List A is stable if A is. The fact that the memory usage of *scary_const* is unbounded is then reflected by the fact that the state of type List A that is needed for *scary_const* is unbounded in size.

Note that we make crucial use of the Fitch-style presentation to rule out *leakyNats*. In the more traditional dual context approach of Krishnaswami [2013], it does not seem possible to have a similar restriction on unfolding of fixed points. The difference is that Simply RaTT "remembers" when we are under a delay whereas that information is lost in the system by Krishnaswami [2013]. In the example of *leakyNats*, the leak stems from the call of *map* in the tail, which in Krishnaswami's system is typed as a regular now judgement, and thus cannot be prevented.

2.3 Function Types

The operational semantics of Simply RaTT uses a heap for delayed computations as well as input streams. The operation delay(t) stores the computation t on the heap and adv retrieves a delayed computation from the heap and evaluates it. In this sense, delay and adv can be understood as computational effects.

Our main result (Theorem 6.3) states that delayed computations and input data on the heap can be safely garbage collected after each computation step. This result relies crucially on the property that open terms typed in now-judgements cannot retrieve delayed computations from the heap. One reason for this is that such terms can not contain adv unless under delay. To maintain this invariant also for function calls, function types $A \to B$ are restricted to functions with no retrieve effects. For this reason, functions may not be constructed in a later-judgement. Later-variables can still be used for case-expressions, and so are included in Simply RaTT. The language could be extended with an extra function type with read effects, constructed by abstracting later-variables in later-judgements, but we found no use for this in our examples.

For example, we can define a function reading an input stream and returning a stream of functions as follows

$$\begin{aligned} &f : \Box\,(\mathsf{Str}\ \mathsf{Nat} \to \mathsf{Str}\ (\mathsf{Nat} \to \mathsf{Nat})) \\ &f = \mathit{map}\ (\mathsf{box}\ (\lambda n\,.\,\lambda x\,.\,n + x)) \end{aligned}$$

and apply streams of functions as follows

$$\begin{aligned} &\mathit{strApp} : \Box\,(\mathsf{Str}\ (A \to B) \to \mathsf{Str}\ A \to \mathsf{Str}\ B) \\ &\mathit{strApp}\,\sharp\,(f :: \mathit{fs})\ (a :: \mathit{as}) = f\ a :: \mathit{strApp} \circledast \mathit{fs} \circledast \mathit{as} \end{aligned}$$

$$
\begin{array}{lll}
\text{Types} & A, B & ::= A \mid 1 \mid \mathsf{Nat} \mid A \times B \mid A + B \mid A \to B \mid \bigcirc A \mid \Box A \mid \mu\alpha.A \\
\text{Values} & v, w & ::= \langle\rangle \mid \bar{n} \mid \lambda x.t \mid \langle v, w\rangle \mid \mathsf{in}_i\, v \mid \mathsf{box}\, t \mid \mathsf{into}\, v \mid \mathsf{fix}\, x.t \mid l \\
\text{Terms} & s, t & ::= \langle\rangle \mid \bar{n} \mid \lambda x.t \mid \langle s, t\rangle \mid \mathsf{in}_i\, t \mid \mathsf{box}\, t \mid \mathsf{into}\, t \mid \mathsf{fix}\, x.t \mid l \mid x \mid t_1\, t_2 \mid t_1 + t_2 \mid \mathsf{adv}\, t \\
& & \quad \mid \mathsf{delay}\, t \mid \mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2 \mid \mathsf{unbox}\, t \mid \mathsf{progress}\, t \mid \mathsf{promote}\, t \mid \mathsf{out}\, t
\end{array}
$$

Fig. 2. Syntax.

Well-formed types $\Theta \vdash A : \mathsf{type}$

$$
\frac{\alpha \in \Theta}{\Theta \vdash \alpha : \mathsf{type}} \qquad \frac{}{\Theta \vdash 1 : \mathsf{type}} \qquad \frac{}{\Theta \vdash \mathsf{Nat} : \mathsf{type}} \qquad \frac{\Theta \vdash A : \mathsf{type} \qquad \Theta \vdash B : \mathsf{type}}{\Theta \vdash A \times B : \mathsf{type}}
$$

$$
\frac{\Theta \vdash A : \mathsf{type} \qquad \Theta \vdash B : \mathsf{type}}{\Theta \vdash A + B : \mathsf{type}} \qquad \frac{\Theta \vdash A : \mathsf{type} \qquad \Theta \vdash B : \mathsf{type}}{\Theta \vdash A \to B : \mathsf{type}} \qquad \frac{\Theta \vdash A : \mathsf{type}}{\Theta \vdash \bigcirc A : \mathsf{type}}
$$

$$
\frac{\Theta \vdash A : \mathsf{type}}{\Theta \vdash \Box A : \mathsf{type}} \qquad \frac{\Theta, \alpha \vdash A : \mathsf{type}}{\Theta \vdash \mu\alpha.A : \mathsf{type}}
$$

Well-formed contexts $\Gamma \vdash$

$$
\frac{}{\emptyset \vdash} \qquad \frac{\Gamma \vdash \qquad \vdash A : \mathsf{type}}{\Gamma, x : A \vdash} \qquad \frac{\Gamma \vdash \qquad \mathsf{token\text{-}free}(\Gamma)}{\Gamma, \sharp \vdash} \qquad \frac{\Gamma \vdash \qquad \mathsf{tick\text{-}free}(\Gamma) \qquad \sharp \in \Gamma}{\Gamma, \checkmark \vdash}
$$

Stable types $A\ \mathsf{stable}$

$$
\frac{}{1\ \mathsf{stable}} \qquad \frac{}{\mathsf{Nat}\ \mathsf{stable}} \qquad \frac{}{\Box A\ \mathsf{stable}} \qquad \frac{A\ \mathsf{stable} \qquad B\ \mathsf{stable}}{A \times B\ \mathsf{stable}} \qquad \frac{A\ \mathsf{stable} \qquad B\ \mathsf{stable}}{A + B\ \mathsf{stable}}
$$

Fig. 3. Context and type formation rules for Simply RaTT.

On the other hand, allowing lambda abstraction of later-variables would type the following (rather contrived) stream definition *leaky*, which breaks the safety of the garbage collection strategy:

$$
\begin{array}{l}
\mathit{leaky}' : \Box\ ((1 \to \mathsf{Bool}) \to \mathsf{Str\ Bool}) \\
\mathit{leaky}'\ \sharp\ p = \mathsf{true} :: \mathsf{delay}\ (\mathsf{adv}\ (\mathbf{if}\ (p\ \langle\rangle)\ \mathbf{then}\ \mathit{leaky}'\ \mathbf{else}\ \mathit{leaky}') \\
\qquad\qquad\qquad\qquad (\lambda x\,.\,\mathsf{head}\ (\mathsf{adv}\ \mathit{leaky}'\ (\lambda y\,.\,\mathsf{true})))) \\
\\
\mathit{leaky} : \Box\ (\mathsf{Str\ Bool}) \\
\mathit{leaky} = \mathsf{box}\ (\mathsf{unbox}\ \mathit{leaky}\ (\lambda x\,.\,\mathsf{true}))
\end{array}
$$

In particular, in the definition of *leaky*′ we use adv on the recursive call of *leaky*′ inside a function. The problem here is that the function body only gets evaluated when applied to an argument. However, this application happens too late – at a time where the recursive call to *leaky*′ has already been garbage collected (cf. Section 3.4).

3 OPERATIONAL SEMANTICS

Following the idea of Krishnaswami [2013], we devise an operational semantics for Simply RaTT that is free of space leaks *by construction*. To this end, the operational semantics is defined in terms

$$\frac{\Gamma, x : A, \Gamma' \vdash \qquad \text{token-free}(\Gamma')}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \langle\rangle : 1} \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash \bar{n} : \mathsf{Nat}} \qquad \frac{\Gamma \vdash s : \mathsf{Nat} \qquad \Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash s + t : \mathsf{Nat}}$$

$$\frac{\Gamma, x : A \vdash t : B \qquad \text{tick-free}(\Gamma)}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash t' : A}{\Gamma \vdash t\, t' : B} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash t' : B}{\Gamma \vdash \langle t, t' \rangle : A \times B}$$

$$\frac{\Gamma \vdash t : A_1 \times A_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \pi_i\, t : A_i} \qquad \frac{\Gamma \vdash t : A_i \qquad i \in \{1, 2\}}{\Gamma \vdash \mathsf{in}_i\, t : A_1 + A_2}$$

$$\frac{\Gamma, x : A_i \vdash t_i : B \qquad \Gamma \vdash t : A_1 + A_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2 : B} \qquad \frac{\Gamma, \checkmark \vdash t : A}{\Gamma \vdash \mathsf{delay}\, t : \bigcirc A}$$

$$\frac{\Gamma \vdash t : \bigcirc A \qquad \Gamma, \checkmark, \Gamma' \vdash}{\Gamma, \checkmark, \Gamma' \vdash \mathsf{adv}\, t : A} \qquad \frac{\Gamma \vdash t : \Box A \qquad \text{token-free}(\Gamma')}{\Gamma, \sharp, \Gamma' \vdash \mathsf{unbox}\, t : A} \qquad \frac{\Gamma, \sharp \vdash t : A}{\Gamma \vdash \mathsf{box}\, t : \Box A}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma, \checkmark, \Gamma' \vdash \qquad A\ \mathsf{stable}}{\Gamma, \checkmark, \Gamma' \vdash \mathsf{progress}\, t : A} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma, \sharp, \Gamma' \vdash \qquad A\ \mathsf{stable}}{\Gamma, \sharp, \Gamma' \vdash \mathsf{promote}\, t : A}$$

$$\frac{\Gamma \vdash t : A[\bigcirc(\mu\alpha.A)/\alpha]}{\Gamma \vdash \mathsf{into}\, t : \mu\alpha.A} \qquad \frac{\Gamma \vdash t : \mu\alpha.A}{\Gamma \vdash \mathsf{out}\, t : A[\bigcirc(\mu\alpha.A)/\alpha]} \qquad \frac{\Gamma, \sharp, x : \bigcirc A \vdash t : A}{\Gamma \vdash \mathsf{fix}\, x.t : \Box A}$$

Fig. 4. Typing rules of Simply RaTT.

of a machine that has access to a store consisting of up to two separate heaps: A 'now' heap η_N from which we can retrieve delayed computations, and a 'later' heap η_L where we can store computations that should be performed in the next time step. Once the machine advances to the next time step, it will delete the 'now' heap η_N and the 'later' heap η_L will become the new 'now' heap. Thus the problem of proving the absence of space leaks is reduced to the problem of soundness, i.e., that well-typed programs never get stuck.

3.1 Term Semantics

The operational semantics of terms is presented in Figure 5. Given a term t together with a store σ, we write $\langle t; \sigma \rangle \Downarrow \langle v; \sigma' \rangle$ to denote that the machine evaluates t in the context of σ to a value v and produces an updated store σ'. Importantly, a store σ can take on three different forms: It may contain no heap, written $\sigma = \bot$; it may consist of one heap η_L, written $\sigma = \sharp\eta_L$; or it may consist of two heaps η_N and η_L, written $\sigma = \sharp\eta_N\checkmark\eta_L$. These different forms of stores enforce effective restrictions on when the machine is allowed to store or retrieve delayed computations. If $\sigma = \bot$, then computations may neither be stored nor retrieved. If $\sigma = \sharp\eta_L$, then computations may be stored in η_L to be retrieved in the next time step. And if $\sigma = \sharp\eta_N\checkmark\eta_L$, computations may be stored in η_L as well as retrieved from η_N. Heaps themselves are simply finite mappings from *heap locations* to terms.

Given a store σ that is not $\bot$, i.e., it is either of the form $\sharp\eta_L$ or $\sharp\eta_N\checkmark\eta_L$, the machine can store delayed computations on the 'later' heap η_L. To this end, we use the notation $\mathsf{later}(\sigma)$ to refer to η_L, and given $l \notin \mathrm{dom}(\eta_L)$, we write $\sigma, l \mapsto t$ for the store $\sharp(\eta_L, l \mapsto t)$ or $\sharp\eta_N\checkmark(\eta_L, l \mapsto t)$,

$$\frac{}{\langle v;\sigma\rangle \Downarrow \langle v;\sigma\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow \langle u;\sigma'\rangle \qquad \langle t';\sigma'\rangle \Downarrow \langle u';\sigma''\rangle}{\langle \langle t,t'\rangle ;\sigma\rangle \Downarrow \langle \langle u,u'\rangle ;\sigma''\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow \langle \langle v_1,v_2\rangle ;\sigma'\rangle \qquad i\in\{1,2\}}{\langle \pi_i(t);\sigma\rangle \Downarrow \langle v_i;\sigma'\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle \qquad i\in\{1,2\}}{\langle \mathsf{in}_i(t);\sigma\rangle \Downarrow \langle \mathsf{in}_i(v);\sigma'\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow \langle \mathsf{in}_i(u);\sigma'\rangle \qquad \langle t_i[v/x];\sigma'\rangle \Downarrow \langle u_i;\sigma''\rangle \qquad i\in\{1,2\}}{\langle \mathsf{case}\ t\ \mathsf{of}\ \mathsf{in}_1\ x.t_1;\mathsf{in}_2\ x.t_2;\sigma\rangle \Downarrow \langle u_i;\sigma''\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow \langle \lambda x.s;\sigma'\rangle \qquad \langle t';\sigma'\rangle \Downarrow \langle v;\sigma''\rangle \qquad \langle s[v/x];\sigma''\rangle \Downarrow \langle v';\sigma'''\rangle}{\langle t\ t';\sigma\rangle \Downarrow \langle v';\sigma'''\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow \langle \overline{m};\sigma'\rangle \qquad \langle t';\sigma'\rangle \Downarrow \langle \overline{n};\sigma''\rangle}{\langle t+t';\sigma\rangle \Downarrow \langle \overline{m+n};\sigma''\rangle} \qquad \frac{\sigma\neq\bot \qquad l=\mathsf{alloc}\,(\sigma)}{\langle \mathsf{delay}\ t;\sigma\rangle \Downarrow \langle l;\sigma,l\mapsto t\rangle}$$

$$\frac{\langle t;\sharp\eta_N\rangle \Downarrow \langle l;\sharp\eta'_N\rangle \qquad \langle \eta'_N(l);\sharp\eta'_N\checkmark\eta_L\rangle \Downarrow \langle v;\sigma'\rangle}{\langle \mathsf{adv}\ t;\sharp\eta_N\checkmark\eta_L\rangle \Downarrow \langle v;\sigma'\rangle} \qquad \frac{\langle t;\bot\rangle \Downarrow \langle v;\bot\rangle \qquad \sigma\neq\bot}{\langle \mathsf{promote}\ t;\sigma\rangle \Downarrow \langle v;\sigma\rangle}$$

$$\frac{\langle t;\sharp\eta_N\rangle \Downarrow \langle v;\sharp\eta'_N\rangle}{\langle \mathsf{progress}\ t;\sharp\eta_N\checkmark\eta_L\rangle \Downarrow \langle v;\sharp\eta'_N\checkmark\eta_L\rangle} \qquad \frac{\langle t;\bot\rangle \Downarrow \langle \mathsf{box}\ t';\bot\rangle \qquad \langle t';\sigma\rangle \Downarrow \langle v;\sigma'\rangle \qquad \sigma\neq\bot}{\langle \mathsf{unbox}\ t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}{\langle \mathsf{into}\ t;\sigma\rangle \Downarrow \langle \mathsf{into}\ v;\sigma'\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow \langle \mathsf{into}\ v;\sigma'\rangle}{\langle \mathsf{out}\ t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}$$

$$\frac{\langle t;\bot\rangle \Downarrow \langle \mathsf{fix}\ x.t';\bot\rangle \qquad \langle t'[l/x];\sigma,l\mapsto \mathsf{unbox}(\mathsf{fix}\ x.t')\rangle \Downarrow \langle v;\sigma'\rangle \qquad \sigma\neq\bot \qquad l=\mathsf{alloc}\,(\sigma)}{\langle \mathsf{unbox}\ t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}$$

Fig. 5. Big-step operational semantics.

$$\frac{\langle t;\sharp\eta\checkmark\rangle \Downarrow \langle v :: l;\sharp\eta_N\checkmark\eta_L\rangle}{\langle t;\eta\rangle \overset{v}{\Longrightarrow} \langle \mathsf{adv}\ l;\eta_L\rangle} \qquad \frac{\langle t;\sharp\eta,l^*\mapsto v :: l^*\checkmark l^*\mapsto\langle\rangle\rangle \Downarrow \langle v' :: l;\sharp\eta_N\checkmark\eta_L,l^*\mapsto\langle\rangle\rangle}{\langle t;\eta\rangle \overset{v/v'}{\Longrightarrow} \langle \mathsf{adv}\ l;\eta_L\rangle}$$

Fig. 6. Small-step operational semantics for stream unfolding and stream processing.

respectively. In turn, $\eta_L, l\mapsto t$ denotes the heap obtained by extending η_L with a new mapping $l\mapsto t$. To allocate a fresh heap locations, we assume a function $\mathsf{alloc}\,(\cdot)$ that takes a store $\sigma\neq\bot$ and returns a heap location l such that $l\notin \mathsf{dom}\,(\mathsf{later}(\sigma))$. That is, given $l=\mathsf{alloc}\,(\sigma)$, we can form the new store $\sigma, l\mapsto t$ without overwriting any mappings that are present in σ.

As the notation suggests, there is a close correspondence between the shape of a context Γ and the shape of a store σ. Terms typable in an initial judgement (cf. Figure 1) can be executed safely with a store $\bot$ – they need not store nor retrieve delayed computations. Terms typable in a now judgement can be executed safely with a store $\sharp\eta_L$ or $\sharp\eta_N\checkmark\eta_L$ – they may store delayed

computations in η_L, but need not retrieve delayed computations. And finally, terms typable in a later judgement can be executed safely in a store of the form $\sharp\eta_N\checkmark\eta_L$ – they may retrieve delayed computations from η_N.

This intuition of the capabilities of the different stores can be observed directly in the semantics for delay and adv: For delay t to evaluate, the machine expects a store that is not $\bot$, i.e., a store $\sharp\eta_L$ or $\sharp\eta_N\checkmark\eta_L$. Then the machine allocates a fresh heap location l in the heap η_L and stores t in it. This corresponds to the fact that delay t can only be typed in a now judgement. Conversely, adv t requires the store to be of the form $\sharp\eta_N\checkmark\eta_L$ so that t can be evaluated safely with the store $\sharp\eta_N$ to a heap location l, which either already existed in η_N or was allocated when evaluating t. In either case, the delayed computation stored at heap location l is retrieved and executed. The combinator progress with its typing rule similar to that of adv, also has similar operational behaviour in terms of how it interacts with the store.

Fixed points are evaluated when a term t that evaluates to a value of the form $\mathsf{fix}\, x.t'$ is unboxed. For a general fixed point combinator, we would expect that $\mathsf{fix}\, x.t'$ unfolds to $t'[\mathsf{fix}\, x.t'/x]$. In our setting, the types dictate that $\mathsf{fix}\, x.t'$ should rather unfold to $t'[\mathsf{delay}(\mathsf{unbox}(\mathsf{fix}\, x.t'))/x]$, because x has type $\bigcirc A$ and $\mathsf{fix}\, x.t'$ has type $\Box A$. This is close to the behaviour of our machine (and would in fact be a safe alternative definition). Instead, however, the machine anticipates that the term allocates a mapping $l \mapsto \mathsf{unbox}(\mathsf{fix}\, x.t')$ on the store and evaluates to that heap location l. Therefore, the machine evaluates the fixed point by allocating a mapping $l \mapsto \mathsf{unbox}(\mathsf{fix}\, x.t')$ on the store right away and evaluating $t'[l/x]$ subsequently.

3.2 Stream Semantics

The careful distinction between a 'now' heap η_N and a 'later' heap η_L is crucial in order to avoid *implicit* space leaks. After the machine has evaluated a term t to a value v and produced a store of the form $\sharp\eta_N\checkmark\eta_L$, we can safely garbage collect the entire heap η_N and compute the next step with the store $\sharp\eta_L\checkmark$. For example, if the original term t was of type Str(Nat), then its value v will be of the form $\overline{n} :: l$, where n is the head of the stream and l is a heap location that points to the delayed computation that computes the tail of the steam. The tail of the stream can then be safely computed by evaluating adv l with the store $\sharp\eta_L\checkmark$, i.e. with the entire 'now' heap η_N garbage collected.

This idea of computing streams is made formal in the definition of the small-step operational semantics $\Longrightarrow$ for streams given in the left half of Figure 6. It starts by evaluating the term to a value of the form $v :: l$, which additionally produces the store $\sharp\eta_N\checkmark\eta_L$. Then the computation can be continued by evaluating adv l in the garbage collected store $\sharp\eta_L\checkmark$, which in turn produces a value $v' :: l'$ and a store $\sharp\eta'_N\checkmark\eta'_L$ – and so on.

Given a closed term t of type $\Box\mathsf{Str}(A)$, we compute the elements $v_1, v_2, v_3, \ldots$ of the stream defined by t as follows:

$$\langle \mathsf{unbox}\, t; \emptyset\rangle \overset{v_1}{\Longrightarrow} \langle t_1; \eta_1\rangle \overset{v_2}{\Longrightarrow} \langle t_2; \eta_2\rangle \overset{v_3}{\Longrightarrow} \ldots$$

where we start with the empty heap $\emptyset$. Each state of the computation $\langle t_i; \eta_i\rangle$ consists of a term t_i and its 'now' heap η_i.

For example, consider the stream $\vdash \mathit{nats} : \Box(\mathsf{Str}(\mathsf{Nat}))$ defined in Section 2.2. To understand how this stream is executed, it is helpful to see how the definition of *nats* desugars to our core calculus. Namely, *nats* is defined as the term $\mathit{from} \boxdot \overline{0}$ where

$$\mathit{from} = \mathsf{fix} f.\lambda n.n :: \mathsf{delay}\,(\mathsf{adv}\, f\, (\mathsf{progress}\,(n + \overline{1})))$$

Here we also unfold the definition of $\odot$. The first three steps of executing the *nats* stream look as follows:

$$\begin{aligned}
\langle \mathsf{unbox}\ \mathit{nats}; \emptyset\rangle &\overset{\overline{0}}{\Longrightarrow} \left\langle \mathsf{adv}\, l_1'; l_1 \mapsto \mathsf{unbox}\ \mathit{from},\ l_1' \mapsto \mathsf{adv}\, l_1\, (\mathsf{progress}\, \overline{0} + \overline{1})\right\rangle \\
&\overset{\overline{1}}{\Longrightarrow} \left\langle \mathsf{adv}\, l_2'; l_2 \mapsto \mathsf{unbox}\ \mathit{from},\ l_2' \mapsto \mathsf{adv}\, l_2\, (\mathsf{progress}\, \overline{1} + \overline{1})\right\rangle \\
&\overset{\overline{2}}{\Longrightarrow} \left\langle \mathsf{adv}\, l_3'; l_3 \mapsto \mathsf{unbox}\ \mathit{from},\ l_3' \mapsto \mathsf{adv}\, l_3\, (\mathsf{progress}\, \overline{2} + \overline{1})\right\rangle \\
&\ \ \vdots
\end{aligned}$$

As expected, the computation produces the consecutive natural numbers. In each step of the computation, the location l_i stores the fixed point *from* that underlies *nats*, whereas l_i' stores the computation that calls that fixed point with the current state of the computation, namely the number $\overline{i}$.

Our main result is that execution of programs by the machine in Figure 5 and Figure 6 is safe. For the stream semantics, this means that we can compute the stream defined by a term t of type $\Box(\mathsf{Str}(A))$ by successive unfolding ad infinitum as follows:

$$\langle \mathsf{unbox}\ t; \emptyset\rangle \overset{v_1}{\Longrightarrow} \langle t_1; \eta_1\rangle \overset{v_2}{\Longrightarrow} \langle t_2; \eta_2\rangle \overset{v_3}{\Longrightarrow} \ldots$$

This intuition is expressed more formally in the following theorem:

Theorem 3.1 (productivity). *Let A be a value type, i.e., a type constructed from* $1, \mathsf{Nat}, +, \times$ *only, and* $\vdash t : \Box(\mathsf{Str}(A))$. *Given any* $n \in \mathbb{N}$, *there is a reduction sequence*

$$\langle \mathsf{unbox}\ t; \emptyset\rangle \overset{v_1}{\Longrightarrow} \langle t_1; \eta_1\rangle \overset{v_2}{\Longrightarrow} \quad \ldots \quad \overset{v_n}{\Longrightarrow} \langle t_n; \eta_n\rangle \quad \textit{such that} \vdash v_i : A \textit{ for all } 1 \le i \le n.$$

3.3 Stream Transducer Semantics

More importantly, our language also facilitates stream processing, that is executing programs of type $\Box(\mathsf{Str}(A) \to \mathsf{Str}(B))$. The small-step operational semantics $\overset{\cdot/\cdot}{\Longrightarrow}$ for executing such programs is given on the right half of Figure 6. So far the store has only been used by the term semantics to store delayed computations. In addition to that purpose, the stream transducer semantics uses the store to transfer the data received from the input steam to the stream transducer. To this end, we assume an arbitrary but fixed heap location l^*, which the machine uses to successively insert the input stream of type $\mathsf{Str}(A)$ as it becomes available. Note that the stream transducer semantics reserves the heap location l^* in new 'later' heap by storing $\langle\rangle$ in it. That means, l^* cannot be allocated by the machine and is available later when the input becomes available and needs to be stored in l^*.

Given a closed term t of type $\Box(\mathsf{Str}(A) \to \mathsf{Str}(B))$, we can execute it as follows:

$$\langle \mathsf{unbox}\ t\ (\mathsf{adv}\, l^*); \emptyset\rangle \overset{v_1/v_1'}{\Longrightarrow} \langle t_1; \eta_1\rangle \overset{v_2/v_2'}{\Longrightarrow} \langle t_2; \eta_2\rangle \overset{v_3/v_3'}{\Longrightarrow} \ldots$$

The machine starts with an empty heap $\emptyset$. In each step $\langle t_i; \eta_i\rangle \overset{v_{i+1}/v_{i+1}'}{\Longrightarrow} \langle t_{i+1}; \eta_{i+1}\rangle$, the machine starts in a state $\langle t_i; \eta_i\rangle$ consisting of a term t_i and heap η_i. Then it reads an input v_{i+1} and subsequently produces the output v_{i+1}' and the next state $\langle t_{i+1}; \eta_{i+1}\rangle$.

Let's consider a simple stream transducer to illustrate the workings of the semantics. The stream transducer *sum* takes a stream of numbers and computes at each point in time the sum of all previous numbers from the input stream. To this end *sum* uses the auxiliary function *sum′* that takes as additional argument the accumulator of type Nat.

$$\begin{aligned}
&\mathit{sum}' : \Box\ (\mathsf{Nat} \to \mathsf{Str}\ (\mathsf{Nat}) \to \mathsf{Str}\ (\mathsf{Nat})) \\
&\mathit{sum}' \sharp\ \mathit{acc}\ (n :: \mathit{ns}) = (\mathit{acc} + n) :: \mathit{sum}' \odot (\mathit{acc} + n) \circledast \mathit{ns}
\end{aligned}$$

$$sum : \Box\,(\mathsf{Str}\,(\mathsf{Nat}) \to \mathsf{Str}\,(\mathsf{Nat}))$$
$$sum = sum' \boxdot \overline{0}$$

To appreciate the workings of the stream transducer semantics, we desugar the definition of sum' in the surface syntax to our core calculus. In addition, we also unfold the definition of $\circledast$:

$$sum' = \mathsf{fix} f.\lambda acc.\lambda s.(acc + \mathsf{head}\, s) :: \mathsf{delay}\,(\mathsf{adv}\,(f \odot (acc + \mathsf{head}\, s))\,(\mathsf{adv}\,(\mathsf{tail}\, s)))$$

Let's look at the first three steps of executing the *sum* stream transducer. To this end, we feed the computation 2, 11, and 5 as input:

$$\begin{array}{ll} & \langle \mathsf{unbox}\ sum; \emptyset\rangle \\ \overset{\overline{2}/\overline{2}}{\Longrightarrow} & \left\langle \mathsf{adv}\, l_1'; l_1 \mapsto \mathsf{unbox}\ sum',\ l_1' \mapsto \mathsf{adv}\,(l_1 \odot (\overline{0} + \mathsf{head}\,(\overline{2} :: l^*)))\,(\mathsf{adv}\,(\mathsf{tail}\,(\overline{2} :: l^*)))\right\rangle \\ \overset{\overline{11}/\overline{13}}{\Longrightarrow} & \left\langle \mathsf{adv}\, l_2'; l_2 \mapsto \mathsf{unbox}\ sum',\ l_2' \mapsto \mathsf{adv}\,(l_2 \odot (\overline{2} + \mathsf{head}\,(\overline{11} :: l^*)))\,(\mathsf{adv}\,(\mathsf{tail}\,(\overline{11} :: l^*)))\right\rangle \\ \overset{\overline{5}/\overline{18}}{\Longrightarrow} & \left\langle \mathsf{adv}\, l_3'; l_3 \mapsto \mathsf{unbox}\ sum',\ l_3' \mapsto \mathsf{adv}\,(l_3 \odot (\overline{13} + \mathsf{head}\,(\overline{5} :: l^*)))\,(\mathsf{adv}\,(\mathsf{tail}\,(\overline{5} :: l^*)))\right\rangle \\ \vdots & \end{array}$$

As expected, we receive 2, 13 (= 2 + 11), and 18 (= 2 + 11 + 5) as result. Moreover, in each step of the computation the location l_i stores the fixed point sum' that underlies the definition of *sum*, whereas l_i' stores the computation that calls that fixed point with the new accumulator value (0 + 2, 2 + 11, and 13 + 5, respectively) and the tail of the input stream.

Corresponding to the productivity property from the previous section, we prove the following causality property that states that the stream transducer semantics never gets stuck. To characterise the causality property, the theorem constructs a family of sets $T_k(A, B)$ which consists of states $\langle t; \eta\rangle$ on which the stream transducer machine can run for k more time steps.

Theorem 3.2 (causality). *Given any value types A and B, there is a family of sets $T_k(A, B)$ such that the following holds for all $k \in \mathbb{N}$:*

(i) If $\vdash t : \Box(\mathsf{Str}(A) \to \mathsf{Str}(B))$ *then* $\langle \mathsf{unbox}\ t\ (\mathsf{adv}\, l^*); \emptyset\rangle \in T_k(A, B)$.

(ii) If $\langle t, \eta\rangle \in T_{k+1}(A, B)$ *and* $\vdash v : A$ *then there are* t', η', *and* $\vdash v' : B$ *such that*

$$\langle t; \eta\rangle \overset{v/v'}{\Longrightarrow} \langle t'; \eta'\rangle \ \text{ and } \ \langle t'; \eta'\rangle \in T_k(A, B).$$

That is, any term t of type $\Box(\mathsf{Str}(A) \to \mathsf{Str}(B))$ defines a causal stream function which is effectively computed by the machine:

$$\langle \mathsf{unbox}\ t; \emptyset\rangle \overset{v_1/v_1'}{\Longrightarrow} \langle t_1; \eta_1\rangle \overset{v_2/v_2'}{\Longrightarrow} \langle t_2; \eta_2\rangle \overset{v_3/v_3'}{\Longrightarrow} \ldots$$

Note that the stream transducer semantics also extends to stream transducers with multiple streams as inputs. This can be achieved by a combinator *split* of type $\Box(\mathsf{Str}(A{\times}B) \to \mathsf{Str}(A){\times}\mathsf{Str}(B))$. Similarly, the semantics also extends to transducers that take an event as input by virtue of a combinator *first* of type $\Box(\mathsf{Str}(1 + A) \to \mathsf{Ev}(A))$. Conversely, transducers producing events instead of streams can be executed using a combinator of type $\Box(\mathsf{Ev}(A) \to \mathsf{Str}(1 + A))$.

We give the proof of Theorem 3.1 and Theorem 3.2 in Section 6. Both results follow from a more general result for the machine, which is formulated using a Kripke logical relation.

3.4 Counterexamples

To conclude this section we review some programs that are rejected by our type system and illustrate their operational behaviour.

Unbox under delay. Recall the alternative definition of the stream of consecutive natural numbers *leakyNats* that uses the *map* combinator. First consider the definition of *leakyNats* in our core calculus:

$$\mathit{leakyNats} = \mathsf{fix}\, s.\overline{0} :: \mathsf{delay}\,(\mathsf{unbox}\,(\mathit{map}\,(\mathsf{box}(\lambda x.x + \overline{1})))) \circledast s$$

Let's contrast the execution of *nats* that we have seen in Section 3.2 with the execution of *leakyNats*:

$$\begin{array}{ll}
 & \langle \mathsf{unbox}\ \mathit{leakyNats}; \emptyset \rangle \\
\overset{\overline{0}}{\Longrightarrow} & \left\langle \mathsf{adv}\, l_1'; \; l_1 \mapsto \mathsf{unbox}\ \mathit{leakyNats},\; l_1' \mapsto \mathsf{unbox}\ \mathit{map}\,(\mathsf{box}\,\lambda x.x + \overline{1})\,(\mathsf{adv}\, l_1) \right\rangle \\
\overset{\overline{1}}{\Longrightarrow} & \left\langle \begin{array}{ll} \mathsf{adv}\, l_2^3; & l_2^0 \mapsto \mathsf{unbox}\ \mathit{leakyNats},\; l_2^1 \mapsto \mathsf{unbox}\ \mathit{map}\,(\mathsf{box}\,\lambda x.x + \overline{1})\,(\mathsf{adv}\, l_2^0), \\ & l_2^2 \mapsto \mathsf{unbox}\ \mathit{step}, \qquad\quad l_2^3 \mapsto \mathsf{adv}\, l_2^2\,(\mathsf{adv}\,(\mathsf{tail}\,(\overline{0} :: l_2^1))) \end{array} \right\rangle \\
\overset{\overline{2}}{\Longrightarrow} & \left\langle \begin{array}{ll} \mathsf{adv}\, l_3^5; & l_3^0 \mapsto \mathsf{unbox}\ \mathit{leakyNats},\; l_3^1 \mapsto \mathsf{unbox}\ \mathit{map}\,(\mathsf{box}\,\lambda x.x + \overline{1})\,(\mathsf{adv}\, l_3^0), \\ & l_3^2 \mapsto \mathsf{unbox}\ \mathit{step}, \qquad\quad l_3^3 \mapsto \mathsf{adv}\, l_3^2\,(\mathsf{adv}\,(\mathsf{tail}\,(\overline{0} :: l_3^1))) \\ & l_3^4 \mapsto \mathsf{unbox}\ \mathit{step}, \qquad\quad l_3^5 \mapsto \mathsf{adv}\, l_3^4\,(\mathsf{adv}\,(\mathsf{tail}\,(\overline{1} :: l_3^3))) \end{array} \right\rangle \\
\vdots &
\end{array}$$

where $\mathit{step} = \mathsf{fix}\, f.\lambda s.\mathsf{unbox}\,(\mathsf{box}\,\lambda n.n + \overline{1})\,(\mathsf{head}\, s) :: (f \circledast \mathsf{tail}\, s)$.

While our type system rejects the term *leakyNats*, a corresponding term is typable in Krishnaswami's calculus [Krishnaswami 2013] and manifests the same memory allocation behaviour as *leakyNats* in our machine.

Lambda abstraction under delay. Recall the definition of the stream *leaky* from Section 2.3. It introduces a lambda abstraction in a later judgement and is therefore rejected by our type system.

$$\begin{array}{ll}
 & \langle \mathsf{unbox}\ \mathit{leaky}; \emptyset \rangle \\
\overset{\mathsf{true}}{\Longrightarrow} & \left\langle \begin{array}{ll} \mathsf{adv}\, l_1'; & l_1 \mapsto \mathsf{unbox}\ \mathit{leaky}', \\ & l_1' \mapsto \mathsf{adv}\,(\mathbf{if}\,(\lambda x.\mathsf{true})\,\langle\rangle\,\mathbf{then}\, l_1\,\mathbf{else}\, l_1)\,(\lambda x.\mathsf{head}\,(\mathsf{adv}\, l_1\,\lambda y.\mathsf{true})) \end{array} \right\rangle \\
\overset{\mathsf{true}}{\Longrightarrow} & \left\langle \begin{array}{ll} \mathsf{adv}\, l_2'; & l_2 \mapsto \mathsf{unbox}\ \mathit{leaky}', \\ & l_2' \mapsto \mathsf{adv}\,(\mathbf{if}\,(\lambda x.\mathsf{head}\,(\mathsf{adv}\, l_1\,\lambda y.\mathsf{true}))\,\langle\rangle\,\mathbf{then}\, l_2\,\mathbf{else}\, l_2) \\ & \qquad\qquad (\lambda x.\mathsf{head}\,(\mathsf{adv}\, l_2\,\lambda y.\mathsf{true})) \end{array} \right\rangle \\
\overset{\cdot}{\not\Longrightarrow} &
\end{array}$$

Note that the term $(\lambda x.\mathsf{head}\,(\mathsf{adv}\, l_1\,\lambda y.\mathsf{true}))$ from the heap after the first step is a value and thus appears unevaluated also in the heap after the second step. However, this term contains a reference to the heap location l_1, which has been garbage collected completing the second step. The machine thus ends up in a stuck state when it tries to dereference the garbage collected heap location l_1 during the third step.

4 GENERIC FRP LIBRARY

This section gives a number of higher-order FRP combinators in Simply RaTT, reminiscent of those found in libraries such as Yampa [Nilsson et al. 2002]. These can be used for programming with streams and events.

Perhaps the simplest example of a stream function is the constant stream over some element. Since we need to output this element in each time step in the future, we require it to come from a stable type. Thus the argument is of type $\Box A$.

$$const : \Box\, A \to \Box\, (\mathsf{Str}\ A)$$
$$const\ a \sharp = \mathsf{unbox}\ a :: const\ a$$

We can now recreate the *zeros* stream presented above as:

$$zeros : \mathsf{Str}\ (\mathsf{Nat})$$
$$zeros = const\ (\mathsf{box}\ 0)$$

Another simple way to generate a stream is to iterate a function $f : A \to A$ over some initial input, such that the output stream will be $(a, fa, f(fa), \ldots)$. Since we will keep using the function at every time step, it needs to be stable, i.e., $f : \Box(A \to A)$. Moreover, since we will keep a state with the current value of type A, that type A must be stable as well. We adopt a syntax like the one used in Haskell for type classes, to denote the additional requirement that a type is stable:

$$iter : A\ \mathsf{stable} \Rightarrow \Box\ (A \to A) \to \Box\ (A \to \mathsf{Str}\ A)$$
$$iter\ f \sharp\, acc = acc :: iter\ f \odot (\mathsf{unbox}\ f\ acc)$$

With this, we can define the stream of natural numbers:

$$nats : \Box\ (\mathsf{Str}\ \mathsf{Nat})$$
$$nats = iter\ (\mathsf{box}\ (\lambda n\,.\,n + 1)) \boxdot 0$$

We may also define a more general *iter* where A need not be stable:

$$iter : \Box\ (A \to \bigcirc\ A) \to \Box\ (A \to \mathsf{Str}\ A)$$
$$iter\ f \sharp\, acc = acc :: iter\ f \circledast (\mathsf{unbox}\ f\ acc)$$

Given some stream, a standard operation in an FRP setting is to *filter* it according to some predicate. This behaviour is easy to implement in Simply RaTT, but because productivity forces us to output a value at each time step, if we take as input $\mathsf{Str}(A)$, we will need to output $\mathsf{Str}(\mathsf{Maybe}(A))$, where $\mathsf{Maybe}(A)$ is a shorthand for $1 + A$. Accordingly, we use the notation nothing and just t to denote $\mathsf{in}_1\ \langle\rangle$ and $\mathsf{in}_2\ t$, respectively.

$$filter : \Box\ (A \to \mathsf{Bool}) \to \Box\ (\mathsf{Str}\ A \to \mathsf{Str}\ (\mathsf{Maybe}\ A))$$
$$filter\ p = map\ \mathsf{box}\ (\lambda a\,.\,\textbf{if}\ \mathsf{unbox}\ p\ a\ \textbf{then}\ \mathsf{just}\ a\ \textbf{else}\ \mathsf{nothing})$$

To go from $\mathsf{Str}(\mathsf{Maybe}(A))$ and back to $\mathsf{Str}(A)$, we can use the *fromMaybe* function that replaces each missing value with a default value:

$$fromMaybe : \Box\ A \to \Box\ (\mathsf{Str}\ (\mathsf{Maybe}\ A) \to \mathsf{Str}\ A)$$
$$fromMaybe\ def \sharp\, (\mathsf{just}\ a \quad :: as) = a \qquad\qquad :: fromMaybe\ def \circledast as$$
$$fromMaybe\ def \sharp\, (\mathsf{nothing} :: as) = \mathsf{unbox}\ def :: fromMaybe\ def \circledast as$$

Given two streams, we can construct the product stream by simply "zipping" the two streams together. It is often easier to construct the more general version where a function is applied to each pair of inputs

$$zipWith : \Box\ (A \to B \to C) \to \Box\ (\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Str}\ C)$$
$$zipWith\ f \sharp\, (a :: as)\ (b :: bs) = \mathsf{unbox}\ f\ a\ b :: zipWith\ f \circledast as \circledast bs$$

The regular zip function is then defined as

$$zip : \Box\ (\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Str}\ (A \times B))$$
$$zip = zipWith\ (\mathsf{box}\ (\lambda a\,.\,\lambda b\,.\,(a, b)))$$

Many applications require the ability to dynamically change the dataflow graph, e.g., when opening and closing windows in a GUI. Such behaviour can be implemented using *switches*, such

as the following, which given an initial stream and a stream event, outputs a stream following the initial stream until it receives a new one on its second argument

$$
\begin{aligned}
&\mathit{switch} : \Box\ (\mathsf{Str}\ A \to \mathsf{Ev}\ (\mathsf{Str}\ A) \to \mathsf{Str}\ A)\\
&\mathit{switch}\ \sharp\ (x :: xs)\ (\mathsf{wait}\ \mathit{fas}) = x :: \mathit{switch} \circledast xs \circledast \mathit{fas}\\
&\mathit{switch}\ \sharp\ xs\quad\ (\mathsf{val}\ \mathit{ys}) \quad = \mathit{ys}
\end{aligned}
$$

As we have described above, we may define streams that require a state, but the state must be defined explicitly. An example is the *scan* function that given a binary operator and an initial state, will output the stream of successive application of the binary operator on the input stream

$$
\begin{aligned}
&\mathit{scan} : B\ \mathsf{stable} \Rightarrow \Box\ (B \to A \to B) \to \Box\ (B \to \mathsf{Str}\ A \to \mathsf{Str}\ B)\\
&\mathit{scan}\ f\ \sharp\ \mathit{acc}\ (a :: as) = \mathit{acc}' :: \mathit{scan}\ f \odot \mathit{acc}' \circledast as\\
&\quad \textbf{where}\ \mathit{acc}' = \mathsf{unbox}\ f\ \mathit{acc}\ a
\end{aligned}
$$

We can now redefine the *sum* function from Section 3.3 as follows:

$$
\begin{aligned}
&\mathit{sum} : \Box\ (\mathsf{Str}\ (\mathsf{Nat}) \to \mathsf{Str}\ (\mathsf{Nat}))\\
&\mathit{sum} = \mathit{scan}\ (\mathsf{box}\ (\lambda n\,.\,\lambda m\,.\,n + m)) \boxdot 0
\end{aligned}
$$

In general, we can encode any computable stream in our language by virtue of the following *unfolding* combinator:

$$
\begin{aligned}
&\mathit{unfold} : \Box\ (X \to A \times \bigcirc\ X) \to \Box\ (X \to \mathsf{Str}\ A)\\
&\mathit{unfold}\ f\ \sharp\ x = \pi_1\ (\mathsf{unbox}\ f\ x) :: \mathit{unfold}\ f \circledast (\pi_2\ (\mathsf{unbox}\ f\ x))
\end{aligned}
$$

To further showcase programming with events, we define the function *await*, which listens for two events and produces a pair event that triggers after both events have arrived. As with *scan*, we need a state to keep the value of the first arriving event while waiting for the second one. This behaviour is implemented by two helper functions, which differ only in which element of the pair is given, and we only show one:

$$
\begin{aligned}
&\mathit{awaitA} : A\ \mathsf{stable} \Rightarrow \Box\ (A \to \mathsf{Ev}\ B \to \mathsf{Ev}\ (A \times B))\\
&\mathit{awaitA}\ \sharp\ a\ (\mathsf{wait}\ \mathit{eb}) = \mathsf{wait}\ (\mathit{awaitA} \odot a \circledast \mathit{eb})\\
&\mathit{awaitA}\ \sharp\ a\ (\mathsf{val}\ b) \quad = \mathsf{val}\ (a, b)
\end{aligned}
$$

We can now define *await* as

$$
\begin{aligned}
&\mathit{await} : A, B\ \mathsf{stable} \Rightarrow \Box\ (\mathsf{Ev}\ A \to \mathsf{Ev}\ B \to \mathsf{Ev}\ (A \times B))\\
&\mathit{await}\ \sharp\ (\mathsf{wait}\ \mathit{ea})\ (\mathsf{wait}\ \mathit{eb}) = \mathit{await} \circledast \mathit{ea} \circledast \mathit{eb}\\
&\mathit{await}\ \sharp\ (\mathsf{val}\ a)\quad \mathit{eb}\qquad\ = \mathsf{unbox}\ \mathit{awaitA}\ a\ \mathit{eb}\\
&\mathit{await}\ \sharp\ \mathit{ea}\qquad\ (\mathsf{val}\ b)\quad = \mathsf{unbox}\ \mathit{awaitB}\ b\ \mathit{ea}
\end{aligned}
$$

The requirement that A and B be stable is crucial since we need to keep the first arriving event until the second occurs.

A second example using events is the *accumulator* combinator. Given a value and a stream of events carrying functions, every time an event is received, the function is applied to the value and output as an event:

$$
\begin{aligned}
&\mathit{accum} : \Box\ A \to \Box\ (\mathsf{Str}\ (\mathsf{Ev}\ (A \to B)) \to \mathsf{Str}\ (\mathsf{Ev}\ B))\\
&\mathit{accum}\ a = \mathit{map}\ (\mathit{eventApp}\ a)
\end{aligned}
$$

The *accum* function uses the helper function below that takes a single event carrying a function and produces an event that applies the function to a given value:

$$\begin{aligned}&\mathit{eventApp} : \Box\, A \to \Box\, (\mathsf{Ev}\ (A \to B) \to \mathsf{Ev}\ B)\\ &\mathit{eventApp}\ a = \mathit{map}\ (\mathsf{box}\ (\lambda f\,.\, f\ (\mathsf{unbox}\ a)))\end{aligned}$$

5 SIMULATING LUSTRE

Lustre is a synchronous dataflow language. Programs describe dataflow graphs and evaluation proceeds in steps, reading input signals and producing output signals. Each signal is associated with a clock, which is always a sub-clock of the global clock, and as such can be described as a sequence of Booleans describing when the clock ticks. A pair of a clock and a signal that produces an output whenever the clock ticks is called a *flow*.

In Simply RaTT clocks can be encoded as Boolean streams and flows as streams of maybe values

$$\mathsf{Clock} = \mathsf{Str}(\mathsf{Bool}) \qquad\qquad \mathsf{Flow}(A) = \mathsf{Str}(\mathsf{Maybe}(A))$$

With these encodings, we now show how to encode some of the basic constructions of Lustre.

The clock associated to a flow ticks whenever the stream produces a value in A. For example, the *basic clock* of the system is the fastest possible clock and the clock *never* is the clock that never ticks. These can be defined as

$$\begin{aligned}&\mathit{basicClock} : \Box\ \mathsf{Clock} && \mathit{never} : \Box\ \mathsf{Clock}\\ &\mathit{basicClock} = \mathit{const}\ (\mathsf{box}\ \mathsf{true}) && \mathit{never} = \mathit{const}\ (\mathsf{box}\ \mathsf{false})\end{aligned}$$

In Simply RaTT, a cycle of the program corresponds to a single stream (transducer) unfolding.

Given a clock, we can slow it down to tick only at certain intervals. We define here a function that given a clock, slows it down to only tick every nth tick. Since we need to carry a state (how often to tick and what step we are at) we define first a helper function:

$$\begin{aligned}&\mathit{everyNthAux} : \mathsf{Nat} \to \Box\ (\mathsf{Nat} \to \mathsf{Clock} \to \mathsf{Clock})\\ &\mathit{everyNthAux}\ \mathit{step}\, \sharp\, \mathit{count}\ (c :: \mathit{cs}) = \mathbf{if}\ (\mathsf{unbox}\ (\mathsf{promote}\ \mathit{step}) = \mathit{count})\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ c \quad\ :: \mathit{everyNthAux}\ \mathit{step} \odot 0 \circledast \mathit{cs}\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathsf{false} :: \mathit{everyNthAux}\ \mathit{step} \odot (\mathit{count} + 1) \circledast \mathit{cs}\end{aligned}$$

We can now define the actual function by giving the helper function an initial state:

$$\begin{aligned}&\mathit{everyNth} : \mathsf{Nat} \to \Box\ (\mathsf{Clock} \to \mathsf{Clock})\\ &\mathit{everyNth}\ n = \mathit{everyNthAux}\ n \boxdot 0\end{aligned}$$

Given a flow and a clock, we can restrict the flow to that clock. If the clock is faster than the "internal" clock of the flow, this will not change the flow.

$$\begin{aligned}&\mathit{when} : \Box\ (\mathsf{Clock} \to \mathsf{Flow}\ A \to \mathsf{Flow}\ A)\\ &\mathit{when}\, \sharp\, (c :: \mathit{cs})\ (a :: \mathit{as}) = (\mathbf{if}\ c\ \mathbf{then}\ a\ \mathbf{else}\ \mathsf{nothing}) :: \mathit{when} \circledast \mathit{cs} \circledast \mathit{as}\end{aligned}$$

Recursive flows are implemented in Lustre using `pre` (previous) and `->` (followed by). We *could* implement these directly in Simply RaTT, but in many Lustre programs `pre` and `->` are used in a pattern that is very natural to Simply RaTT programs: `pre` is used to keep track of some state (that we may update) and `->` provides the initial state. As an example, consider the Lustre node that computes the flow of natural numbers:

```
n = 0 -> pre(n) + 1
```

The equivalent Simply RaTT program is

$$\begin{aligned}&\mathit{nats} : \Box\ (\mathsf{Flow}\ (\mathsf{Nat}))\\ &\mathit{nats} = \mathit{natsAux} \boxdot 0\end{aligned}$$

$$
\begin{aligned}
&\textbf{where}\ \mathit{natsAux} : \Box\ (\mathsf{Nat} \to \mathsf{Flow}\ (\mathsf{Nat})) \\
&\qquad \mathit{natsAux} \sharp \mathit{pre} = \mathsf{just}\ \mathit{pre} :: \mathit{natsAux} \odot (\mathit{pre} + 1)
\end{aligned}
$$

which is similarly composed of an initial state and then the actual computation, which may refer to the previous value.

As another example, consider the following Lustre node which takes a Boolean flow b as input:

```
edge = false -> (b and not pre(b))
```

This output flow is true when it detects a "rising edge" in its input flow, i.e., when the input goes from false to true. This is translated in a similar way to a helper function that does the computation:

$$
\begin{aligned}
&\mathit{edgeAux} : \Box\ (\mathsf{Bool} \to \mathsf{Flow}\ (\mathsf{Bool}) \to \mathsf{Flow}\ (\mathsf{Bool})) \\
&\mathit{edgeAux} \sharp \mathit{pre}\ (\mathsf{just}\ b \quad :: bs) = b' \quad :: \mathit{edgeAux} \odot b' \circledast cs \\
&\mathit{edgeAux} \sharp \mathit{pre}\ (\mathsf{nothing} :: bs) = \mathit{pre} :: \mathit{edgeAux} \odot \mathit{pre} \circledast cs \\
&\quad \textbf{where}\ b' = (b\ \mathsf{and}\ (\neg\ \mathit{pre}))
\end{aligned}
$$

and then a function giving the initial state:

$$
\begin{aligned}
&\mathit{edge} : \Box\ (\mathsf{Flow}\ (\mathsf{Bool}) \to \mathsf{Flow}\ (\mathsf{Bool})) \\
&\mathit{edge} = \mathit{edgeAux} \boxdot \mathsf{false}
\end{aligned}
$$

Since a flow is restricted to its internal clock, it may not produce anything at many ticks of the basic clock. To alleviate this, Lustre provides the `current` operator, which given a flow on a clock slower than the basic clock, fills the holes in the flow with whatever the latest values was. The equivalent Simply RaTT program is:

$$
\begin{aligned}
&\mathit{current} : A\ \mathsf{stable} \Rightarrow \Box\ (\mathsf{Flow}\ A \to \mathsf{Flow}\ A) \\
&\mathit{current}\ as = \mathsf{box}\ (\lambda as\,.\,\mathsf{unbox}\ \mathit{currentAux}\ (\mathsf{head}\ as)\ as) \\
&\quad \textbf{where}\ \mathit{currentAux} : A\ \mathsf{stable} \Rightarrow \Box\ (\mathsf{Maybe}\ A \to \mathsf{Flow}\ A \to \mathsf{Flow}\ A) \\
&\qquad\qquad \mathit{currentAux} \sharp \mathit{pre}\ (\mathsf{just}\ a \quad :: as) = \mathsf{just}\ a :: \mathit{currentAux} \odot \mathsf{just}\ a \circledast as \\
&\qquad\qquad \mathit{currentAux} \sharp \mathit{pre}\ (\mathsf{nothing} :: as) = \mathit{pre} \quad :: \mathit{currentAux} \odot \mathit{pre} \circledast as
\end{aligned}
$$

Our last example is an implementation of `counter` from the Lustre V6 manual [Erwan Jahier and Halbwachs 2019]. The `counter` program takes as input an initial value and a constant that determines how much to increment in each step. Its current state is stored in *pre*. It then listens to two flows, an "increment" flow and a "reset" flow. If the counter receives true on the increment flow, it increments the counter by the increment constant. If it receives true on the reset flow, it resets the counter to the initial value and otherwise it will continue in the same state.

$$
\begin{aligned}
&\mathit{counter} : \mathsf{Nat} \to \mathsf{Nat} \to \Box\ (\mathsf{Nat} \to \mathsf{Flow\ Bool} \to \mathsf{Flow\ Bool} \to \mathsf{Flow\ Nat}) \\
&\mathit{counter}\ \mathit{init}\ \mathit{incr} \sharp \mathit{pre}\ (\mathsf{just}\ s \quad :: ss)\ (\mathsf{just}\ r \quad :: rs) = \\
&\quad \textbf{if}\ s\ \textbf{then}\ \mathsf{just}\ \mathit{init} \qquad :: \mathit{counter}\ \mathit{init}\ \mathit{incr} \odot \mathit{init} \circledast ss \circledast rs \\
&\quad \textbf{else}\ \mathsf{just}\ (\mathit{pre} + \mathit{incr}) \qquad :: \mathit{counter}\ \mathit{init}\ \mathit{incr} \odot (\mathit{pre} + \mathit{incr}) \circledast ss \circledast rs \\
&\mathit{counter}\ \mathit{init}\ \mathit{incr} \sharp \mathit{pre}\ (\mathsf{nothing} :: ss)\ (\mathsf{just}\ r \quad :: rs) = \\
&\quad \textbf{if}\ r\ \textbf{then}\ \mathsf{just}\ \mathit{init} \qquad :: \mathit{counter}\ \mathit{init}\ \mathit{incr} \odot \mathit{init} \circledast ss \circledast rs \\
&\quad \textbf{else}\ \mathsf{just}\ \mathit{pre} \qquad :: \mathit{counter}\ \mathit{init}\ \mathit{incr} \odot \mathit{pre} \circledast ss \circledast rs \\
&\mathit{counter}\ \mathit{init}\ \mathit{incr} \sharp \mathit{pre}\ (\mathsf{just}\ s \quad :: ss)\ (\mathsf{nothing} :: rs) = \\
&\quad \textbf{if}\ r\ \textbf{then}\ \mathsf{just}\ (\mathit{pre} + \mathit{incr}) :: \mathit{counter}\ \mathit{init}\ \mathit{incr} \odot (\mathit{pre} + \mathit{incr}) \circledast ss \circledast rs \\
&\quad \textbf{else}\ \mathsf{just}\ \mathit{pre} \qquad :: \mathit{counter}\ \mathit{init}\ \mathit{incr} \odot \mathit{pre} \circledast ss \circledast rs \\
&\mathit{counter}\ \mathit{init}\ \mathit{incr} \sharp \mathit{pre}\ (\mathsf{nothing} :: ss)\ (\mathsf{nothing} :: rs) = \\
&\quad \mathsf{just}\ \mathit{pre} \qquad :: \mathit{counter}\ \mathit{init}\ \mathit{incr} \odot \mathit{pre} \circledast ss \circledast rs
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{V}[\![\mathsf{Nat}]\!]^{\overline{H}}_{\sigma} &= \{\overline{n} \mid n \in \mathbb{N}\} \\
\mathcal{V}[\![1]\!]^{\overline{H}}_{\sigma} &= \{\langle\rangle\} \\
\mathcal{V}[\![A \times B]\!]^{\overline{H}}_{\sigma} &= \left\{ \langle v_1, v_2\rangle \,\middle|\, v_1 \in \mathcal{V}[\![A]\!]^{\overline{H}}_{\sigma} \wedge v_2 \in \mathcal{V}[\![B]\!]^{\overline{H}}_{\sigma} \right\} \\
\mathcal{V}[\![A + B]\!]^{\overline{H}}_{\sigma} &= \left\{ \mathsf{in}_1\, v \,\middle|\, v \in \mathcal{V}[\![A]\!]^{\overline{H}}_{\sigma} \right\} \cup \{\mathsf{in}_2\, v \mid v \in \mathcal{V}[\![B]\!]^{\overline{H}}_{\sigma}\} \\
\mathcal{V}[\![A \to B]\!]^{\overline{H}}_{\sigma} &= \left\{ \lambda x.t \,\middle|\, \forall(\sigma' \diamond \overline{H}') \sqsupseteq (\mathsf{gc}(\sigma) \diamond \overline{H}).\forall w \in \mathcal{V}[\![A]\!]^{\overline{H}'}_{\sigma'}.t[w/x] \in \mathcal{T}[\![B]\!]^{\overline{H}'}_{\sigma'} \right\} \\
\mathcal{V}[\![\Box A]\!]^{\overline{H}}_{\sigma} &= \left\{ v \,\middle|\, \forall \overline{H}' \leq_{\mathsf{suf}} \overline{H}.\mathsf{unbox}(v) \in \mathcal{T}[\![A]\!]^{\overline{H}'}_{\sharp} \right\} \\
\mathcal{V}[\![\bigcirc A]\!]^{()}_{\sigma} &= \{l \mid l \text{ is any heap location}\} \\
\mathcal{V}[\![\bigcirc A]\!]^{H;\overline{H}}_{\sigma} &= \left\{ l \,\middle|\, \forall \eta \in H.\sigma(l) \in \mathcal{T}[\![A]\!]^{\overline{H}}_{\mathsf{gc}(\sigma)\checkmark\eta} \right\} \\
\mathcal{V}[\![\mu\alpha.A]\!]^{\overline{H}}_{\sigma} &= \left\{ \mathsf{into}(v) \,\middle|\, v \in \mathcal{V}[\![A[\bigcirc(\mu\alpha.A)/\alpha]]\!]^{\overline{H}}_{\sigma} \right\}
\end{aligned}
$$

$$
\mathcal{T}[\![A]\!]^{\overline{H}}_{\sigma} = \left\{ t \,\middle|\, \forall(\sigma' \diamond \overline{H}') \sqsupseteq_{\checkmark} (\sigma \diamond \overline{H}).\exists \sigma'', v.\, \langle t; \sigma'\rangle \Downarrow \langle v; \sigma''\rangle \wedge v \in \mathcal{V}[\![A]\!]^{\overline{H}'}_{\sigma''} \right\}
$$

$$
\begin{aligned}
C[\![\cdot]\!]^{\overline{H}}_{\perp} &= \{\star\} \\
C[\![\Gamma, x : A]\!]^{\overline{H}}_{\sigma} &= \left\{ \gamma[x \mapsto v] \,\middle|\, \gamma \in C[\![\Gamma]\!]^{\overline{H}}_{\sigma}, v \in \mathcal{V}[\![A]\!]^{\overline{H}}_{\sigma} \right\} \\
C[\![\Gamma, \checkmark]\!]^{\overline{H}}_{\sharp\eta_N\checkmark\eta_L} &= C[\![\Gamma]\!]^{\{\eta_L\},\overline{H}}_{\sharp\eta_N} \\
C[\![\Gamma, \sharp]\!]^{\overline{H}}_{\sigma} &= \bigcup_{\overline{H} \leq_{\mathsf{suf}} \overline{H}'} C[\![\Gamma]\!]^{\overline{H}'}_{\perp} \qquad \text{if } \sigma \neq \perp
\end{aligned}
$$

Garbage Collection:

$$
\begin{aligned}
\mathsf{gc}(\perp) &= \perp \\
\mathsf{gc}(\sharp\eta_L) &= \sharp\eta_L \\
\mathsf{gc}(\sharp\eta_N\checkmark\eta_L) &= \sharp\eta_L
\end{aligned}
$$

Fig. 7. Logical Relation.

6 METATHEORY

Since the operational semantics rules out space leaks by construction, it only remains to be shown that the type system is sound, i.e., well-typed terms never get stuck. To this end, we devise a Kripke logical relation. Essentially, such a logical relation is a family $[\![A]\!]_w$ of sets of closed terms that satisfy the desired soundness property. This family of sets is indexed by w drawn from a suitable set of 'worlds' and is defined inductively on the structure of the type A and w. Then the proof of soundness is reduced to a proof that $\vdash t : A$ implies $t \in [\![A]\!]_w$ for all possible worlds. Finally we show how this soundness result is used to prove Theorem 3.1 and Theorem 3.2. All results have been formalised in the accompanying Coq proofs.

6.1 Worlds

To a first approximation, the worlds in our logical relation contain two components: a store σ and a number n, written $[\![A]\!]^n_\sigma$. The number index n allows us to define the logical relation for recursive types via step-indexing [Appel and McAllester 2001]. Concretely, this is achieved by defining $[\![\bigcirc A]\!]^{n+1}_\sigma$ in terms of $[\![A]\!]^n_\sigma$. Since unfolding recursive types $\mu\alpha.A$ to $A[\bigcirc\mu\alpha.A/\alpha]$ introduces a $\bigcirc$ modality, we thus achieve that the step index n decreases for recursive types. In essence, this means that terms in the logical relation $[\![A]\!]^n_\sigma$ can be executed safely for the next n time steps starting with

the store σ. Ultimately, the index σ enables us to prove that the garbage collection performed by the stream and stream transducer semantics (cf. Figure 6) is sound.

While this setup would be sufficient to prove soundness for the stream semantics (Theorem 3.1), it is not enough for the soundness of the stream transducer semantics (Theorem 3.2): To characterise our soundness property it is not enough to require that a term can be executed n more time steps. We also need to know what the input to a stream transducer of type $\mathrm{Str}(A) \to \mathrm{Str}(B)$ looks like, namely a stream where the first n elements are values of type A. We achieve this by describing what the heaps should look like in the next n time steps. Concretely, we assume a finite sequence $(H_1; \ldots; H_n)$, where each H_i is the set of heaps that we could potentially encounter i time steps into the future.

To summarise, the worlds in our logical relation consist of a store σ and a finite sequence $(H_1; \ldots; H_n)$ of sets of heaps. Instead of, $(H_1; \ldots; H_n)$ we also write $\overline{H}$, and we use the notation $(\sigma \diamond \overline{H})$ to refer to the world consisting of the store σ and the sequence $\overline{H}$. Intuitively, σ is the store for which the term in the logical relation can be safely executed, whereas each H_i in $\overline{H}$ contains all heaps for which the term can be safely executed after i time steps have passed.

A crucial ingredient of a Kripke logical relation is a preorder $\lesssim$ on the set of worlds such that the logical relation is closed under that preorder in the sense that $w \lesssim w'$ implies $[\![A]\!]_w \subseteq [\![A]\!]_{w'}$. To this end, we use a partial order $\sqsubseteq$ on heaps, which is the standard partial order on partial maps, i.e., $\eta \sqsubseteq \eta'$ iff $\eta(l) = \eta'(l)$ for all $l \in \mathrm{dom}(\eta)$. Moreover, we extend this order to stores in two different ways, resulting in the two orders $\sqsubseteq$ and $\sqsubseteq_\checkmark$:

$$\frac{}{\bot \sqsubseteq \bot} \qquad \frac{\eta \sqsubseteq \eta'}{\sharp\eta \sqsubseteq \sharp\eta'} \qquad \frac{\eta_N \sqsubseteq \eta'_N \qquad \eta_L \sqsubseteq \eta'_L}{\sharp\eta_N \checkmark \eta_L \sqsubseteq \sharp\eta'_N \checkmark \eta'_L} \qquad \frac{\sigma \sqsubseteq \sigma'}{\sigma \sqsubseteq_\checkmark \sigma'} \qquad \frac{\eta \sqsubseteq \eta'}{\sharp\eta \sqsubseteq_\checkmark \sharp\eta'' \checkmark \eta'}$$

That is, the heap order $\sqsubseteq$ is lifted to stores pointwise, whereas $\sqsubseteq_\checkmark$ extends $\sqsubseteq$ by defining $\sharp\eta_L \sqsubseteq_\checkmark \sharp\eta_N \checkmark \eta_L$. The more general order $\sqsubseteq_\checkmark$ is used in the logical relation, whereas the more restrictive $\sqsubseteq$ is needed to characterise the following property of the operational semantics:

Lemma 6.1. *Given any term t, value v, and pair of stores σ, σ' such that $\langle t; \sigma\rangle \Downarrow \langle v; \sigma'\rangle$, then $\sigma \sqsubseteq \sigma'$.*

We also extend the heap order $\sqsubseteq$ to sets of heaps and sequences of sets of heaps:

$$\begin{array}{rcll} H & \sqsubseteq & H' & \iff \forall \eta' \in H'. \exists \eta \in H. \eta \sqsubseteq \eta' \\ (H_1; H_2; \ldots; H_n) & \sqsubseteq & (H'_1; H'_2; \ldots; H'_n) & \iff \forall 1 \le i \le n. H_i \sqsubseteq H'_i \end{array}$$

That is, if $H \sqsubseteq H'$, then for every heap in H' there is a smaller one in H, and this ordering is lifted pointwise to finite sequences.

Finally, we combine these orderings to worlds pointwise as well

$$\begin{array}{rcll} (\sigma \diamond \overline{H}) & \sqsubseteq & (\sigma' \diamond \overline{H}') & \iff \sigma \sqsubseteq \sigma' \quad \wedge \quad \overline{H} \sqsubseteq \overline{H}' \\ (\sigma \diamond \overline{H}) & \sqsubseteq_\checkmark & (\sigma' \diamond \overline{H}') & \iff \sigma \sqsubseteq_\checkmark \sigma' \quad \wedge \quad \overline{H} \sqsubseteq \overline{H}' \end{array}$$

Note that whenever $\overline{H} \sqsubseteq \overline{H}'$, then $\overline{H}$ and $\overline{H}'$ are of the same length. That is, both $\overline{H}$ and $\overline{H}'$ describe the same number of future time steps. In order to describe a possible future world, i.e., after some time steps have passed, we use suffix ordering $\le_{\mathsf{suf}}$ on sequences. We say that $\overline{H}$ is a suffix of $\overline{H}'$, written $\overline{H} \le_{\mathsf{suf}} \overline{H}'$ iff there is a sequence $\overline{H}''$ such that H' is equal to the concatenation of $\overline{H}''$ and $\overline{H}$, written $\overline{H}''; \overline{H}$. Thus, a suffix $\overline{H} \le_{\mathsf{suf}} \overline{H}'$ describes a future state where the prefix $\overline{H}''$ has already been consumed.

6.2 Logical Relation

Our logical relation consists of two parts: A *value relation* $\mathcal{V}[\![A]\!]_w$ that contains all values that semantically inhabit type A at the world w, and a corresponding *term relation* $\mathcal{T}[\![A]\!]_w$ containing terms. Given a world $(\sigma \diamond \overline{H})$ we write $\mathcal{V}[\![A]\!]^{\overline{H}}_{\sigma}$ and $\mathcal{T}[\![A]\!]^{\overline{H}}_{\sigma}$ instead of $\mathcal{V}[\![A]\!]_{(\sigma \diamond \overline{H})}$ and $\mathcal{T}[\![A]\!]_{(\sigma \diamond \overline{H})}$, respectively. The two relations are defined by mutual induction in Figure 7. More precisely, the two relations are defined by well-founded induction by the lexicographic ordering on the triple $(|\overline{H}|, |A|, e)$, where $|\overline{H}|$ is the length of $\overline{H}$, $|A|$ is the size of A defined below, and $e = 1$ for the term relation and $e = 0$ for the value relation.

$$\begin{aligned} |\alpha| = |\bigcirc A| = |\mathsf{Nat}| = |1| &= 1 \\ |A \times B| = |A + B| = |A \to B| &= 1 + |A| + |B| \\ |\Box A| = |\mu\alpha.A| &= 1 + |A| \end{aligned}$$

We define the size of $\bigcirc A$ to be the same as α. Thus $A[\bigcirc \mu\alpha.A/\alpha]$ is strictly smaller than $\mu\alpha.A$. This justifies the well-foundedness for recursive types. For types $\bigcirc A$, the well-foundedness of the definition can be observed by the fact that $\overline{H}$ is strictly shorter than $H; \overline{H}$, which is a shorthand notation for the sequence $(H; H_1; \ldots; H_n)$ where $\overline{H} = (H_1; \ldots; H_n)$.

The definition of the value relation for $\langle\rangle$, Nat, $\times$, and $+$ is standard. The definition of $\mathcal{V}[\![\Box A]\!]^{\overline{H}}_{\sigma}$ expresses the fact that all its inhabitants can be evaluated safely at any time in the future. To express this, we use suffix ordering $\leq_{\mathsf{suf}}$. A value in $\mathcal{V}[\![\Box A]\!]^{\overline{H}}_{\sigma}$ may be unboxed and subsequently evaluated at any time in the future, i.e., in the context of any suffix of $\overline{H}$.

The value relation for types $\bigcirc A$ encapsulates the soundness of garbage collection. The set $\mathcal{V}[\![\bigcirc A]\!]^{H;\overline{H}}_{\sigma}$ contains all heap locations that point to terms that can be executed safely in the next time step. The notation $\sigma(l)$ is a shorthand for $\eta_L(l)$ given that $\sigma = \sharp\eta_L$ or $\sigma = \sharp\eta_N \checkmark \eta_L$. Hence, we look up the location l in the 'later' heap of σ and require that the term that we find can be executed with the store obtained from σ by first garbage collecting the 'now' heap (if present) and extending it with any future heap drawn from H.

Garbage collection is also crucial in the definition of $\mathcal{V}[\![A \to B]\!]^{\overline{H}}_{\sigma}$, which only contains lambda abstractions that can be applied in a garbage collected store. This reflects the restriction of the typing rule for lambda abstraction, which requires the context Γ to be tick-free. The *leaky* example in Section 3.4 illustrates the necessity of this restriction. Semantically, this implies the following essential property of values:

Lemma 6.2. *For all $A, \sigma, \overline{H}$, we have that $\mathcal{V}[\![A]\!]^{\overline{H}}_{\sigma} \subseteq \mathcal{V}[\![A]\!]^{\overline{H}}_{\mathrm{gc}(\sigma)}$.*

That is, after evaluating a term to a value, we can safely garbage collect the 'now' heap.

Finally, we obtain the soundness of the language by the following fundamental property of the logical relation $\mathcal{T}[\![A]\!]^{\overline{H}}_{\sigma}$.

Theorem 6.3 (Fundamental Property). *Given $\Gamma \vdash t : A$, and $\gamma \in C[\![\Gamma]\!]^{\overline{H}}_{\sigma}$, then $t\gamma \in \mathcal{T}[\![A]\!]^{\overline{H}}_{\sigma}$.*

The theorem is proved by a lengthy but entirely standard induction on the typing relation $\Gamma \vdash t : A$. Two crucial ingredients to the proof are that all logical relations are closed under the ordering $\sqsubseteq_{\checkmark}$ on worlds, and that $C[\![\Gamma]\!]^{\overline{H}}_{\sigma}$ captures the correspondence between the tokens occurring in Γ and σ, namely they have the same number of locks and σ may not have fewer ticks than Γ.

6.3 Soundness of Stream and Stream Transducer Semantics

We conclude this section by demonstrating how we can use the fundamental property of our logical relation for proving the soundness of the abstract machines for evaluating streams (Theorem 3.1)

and stream transducers (Theorem 3.2), which amounts to proving productivity and causality of the calculus.

First, we observe that the operational semantics is deterministic:

Proposition 6.4 (deterministic machine).
(1) If $\langle t;\sigma\rangle \Downarrow \langle v_1;\sigma_1\rangle$ *and* $\langle t;\sigma\rangle \Downarrow \langle v_2;\sigma_2\rangle$*, then* $v_1 = v_2$ *and* $\sigma_1 = \sigma_2$*.*
(2) If $\langle t;\eta\rangle \overset{v_1}{\Longrightarrow} \langle t_1;\eta_1\rangle$ *and* $\langle t;\eta\rangle \overset{v_2}{\Longrightarrow} \langle t_2;\eta_2\rangle$*, then* $v_1 = v_2$*,* $t_1 = t_2$*, and* $\eta_1 = \eta_2$*.*
(3) If $\langle t;\eta\rangle \overset{v/v_1}{\Longrightarrow} \langle t_1;\eta_1\rangle$ *and* $\langle t;\eta\rangle \overset{v/v_2}{\Longrightarrow} \langle t_2;\eta_2\rangle$*, then* $v_1 = v_2$*,* $t_1 = t_2$*, and* $\eta_1 = \eta_2$*.*

Before we can prove Theorem 3.1, we need the following property of value types, i.e., types constructed from 1, Nat, +, ×

Lemma 6.5. *Let* A *be a value type and* $(\sigma \diamond \overline{H})$ *a world.*
(i) For all values v*, we have that* $v \in \mathcal{V}[\![A]\!]^{\overline{H}}_{\sigma}$ *iff* $\vdash v : A$*.*
(ii) $\mathcal{V}[\![A]\!]^{\overline{H}}_{\sigma}$ *is non-empty.*

Proof. By a straightforward induction on A. □

For the proof of Theorem 3.1, we construct for each type A the following family of sets $S_k(A)$, which intuitively contains all states on which the stream semantics can run for k more steps:

$$S_k(A) = \left\{ \langle t;\eta\rangle \;\middle|\; t \in \mathcal{T}[\![\mathsf{Str}(A)]\!]^{\{\emptyset\}^k}_{\sharp\eta\checkmark} \right\}$$

where $\{\emptyset\}$ is the singleton set containing the empty heap, and $\{\emptyset\}^k$ is the sequence containing k copies of $\{\emptyset\}$. Theorem 3.1 follows from the following lemma and the fact that the operational semantics is deterministic:

Lemma 6.6 (productivity). *Given any value type* A*, the following holds for all* $k \in \mathbb{N}$*:*
(i) If $\vdash t : \Box(\mathsf{Str}(A))$ *then* $\langle \mathsf{unbox}\ t;\emptyset\rangle \in S_k(A)$*.*
(ii) If $\langle t,\eta\rangle \in S_{k+1}(A)$ *then there are* t'*,* η'*, and* $\vdash v : A$ *such that*

$$\langle t;\eta\rangle \overset{v}{\Longrightarrow} \langle t';\eta'\rangle \;\text{ and }\; \langle t';\eta'\rangle \in S_k(A).$$

Proof.
(i) $\vdash t : \Box(\mathsf{Str}(A))$ implies $\sharp \vdash \mathsf{unbox}\ t : \mathsf{Str}(A)$ which by Theorem 6.3, implies that $\mathsf{unbox}\ t \in \mathcal{T}[\![\mathsf{Str}(A)]\!]^{\{\emptyset\}^k}_{\sharp}$ and thus also $\mathsf{unbox}\ t \in \mathcal{T}[\![\mathsf{Str}(A)]\!]^{\{\emptyset\}^k}_{\sharp\checkmark}$. Hence $\langle \mathsf{unbox}\ t;\emptyset\rangle \in S_k(A)$.
(ii) Let $\langle t,\eta\rangle \in S_{k+1}(A)$. Then $t \in \mathcal{T}[\![\mathsf{Str}(A)]\!]^{\{\emptyset\}^{k+1}}_{\sharp\eta\checkmark}$, which means that $\langle t;\sharp\eta\checkmark\rangle \Downarrow \langle w;\sigma\rangle$ and $w \in \mathcal{V}[\![\mathsf{Str}(A)]\!]^{\{\emptyset\}^{k+1}}_{\sigma}$. Hence, $w = v :: l$ with $v \in \mathcal{V}[\![A]\!]^{\{\emptyset\}^{k+1}}_{\sigma}$ and $l \in \mathcal{V}[\![\bigcirc\mathsf{Str}(A)]\!]^{\{\emptyset\}^{k+1}}_{\sigma}$. Moreover, by Lemma 6.5 $\vdash v : A$ and by Lemma 6.1 $\sigma = \sharp\eta_N\checkmark\eta_L$. Hence, $\mathsf{adv}\, l \in \mathcal{T}[\![\mathsf{Str}(A)]\!]^{\{\emptyset\}^k}_{\sharp\eta_L\checkmark}$. That is, $\langle t;\eta\rangle \overset{v}{\Longrightarrow} \langle \mathsf{adv}\, l;\eta_L\rangle$ and $\langle \mathsf{adv}\, l;\eta_L\rangle \in T_k(A)$. □

For the proof of causality we need the following property of the operational semantics, which essentially states that we never read from the 'later' heap.

Lemma 6.7. *If* $\langle t;\sigma, l \mapsto u\rangle \Downarrow \langle v;\sigma', l \mapsto u\rangle$*, then* $\langle t;\sigma, l \mapsto u'\rangle \Downarrow \langle v;\sigma', l \mapsto u'\rangle$ *for any* u'*.*

To prove the above lemma we have to make the following reasonable assumption about the function $\mathsf{alloc}\,(\cdot)$ that performs the allocation of fresh heap locations: Given two stores σ, σ' with $\mathsf{dom}\,(\mathsf{later}(\sigma)) = \mathsf{dom}\,(\mathsf{later}(\sigma'))$, we have that $\mathsf{alloc}\,(\sigma) = \mathsf{alloc}\,(\sigma')$. In other words, $\mathsf{alloc}\,(\sigma)$ only depends on the domain of the 'later' heap. For example, if the set of heap locations is just $\mathbb{N}$, then $\mathsf{alloc}\,(\sigma)$ could be implemented as the smallest heap location that is fresh for the 'later' heap of σ.

Analogously to the family of sets $S_k(A)$, we construct a family of sets $T_k(A,B)$ that contains all states which are safe to run for k more steps on the stream transducer semantics:

$$T_k(A,B) = \left\{ \langle t, \eta \rangle \;\middle|\; l^* \notin \mathrm{dom}\,(\eta) \wedge \forall v, w \in \mathcal{V}[\![A]\!]^{()}_{\bot}. t \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^k(A)}_{\sharp\eta, l^* \mapsto v :: l^* \checkmark l^* \mapsto w :: l^*} \right\}$$

where $H(A) = \left\{ l^* \mapsto v :: l^* \;\middle|\; v \in \mathcal{V}[\![A]\!]^{()}_{\bot} \right\}$ and $H^k(A)$ is the sequence of k copies of $H(A)$.

Finally, we give the proof of causality:

Proof of Theorem 3.2.

(i) Given $\vdash t : \Box(\mathrm{Str}(A) \to \mathrm{Str}(B))$ and $v, v' \in \mathcal{V}[\![A]\!]^{()}_{\bot}$, we need to show that $\mathrm{unbox}\, t\, (\mathrm{adv}\, l^*) \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H(A)^k}_{\sharp l^* \mapsto v :: l^* \checkmark l^* \mapsto v' :: l^*}$. By induction on $k+1$ we can show that $l^* \in \mathcal{V}[\![\bigcirc \mathrm{Str}(A)]\!]^{H^{k+1}(A)}_{\sharp l^* \mapsto v :: l^*}$. By definition of the value relation, this means that $v :: l^* \in \mathcal{T}[\![\mathrm{Str}(A)]\!]^{H^k(A)}_{\sharp l^* \mapsto v :: l^* \checkmark l^* \mapsto v' :: l^*}$, which in turn implies that $\mathrm{adv}\, l^* \in \mathcal{T}[\![\mathrm{Str}(A)]\!]^{H^k(A)}_{\sharp l^* \mapsto v :: l^* \checkmark l^* \mapsto v' :: l^*}$. Since $\vdash t : \Box(\mathrm{Str}(A) \to \mathrm{Str}(B))$, we know that $\sharp \vdash \mathrm{unbox}\, t : \mathrm{Str}(A) \to \mathrm{Str}(B)$. Using Theorem 6.3 we thus obtain that $\mathrm{unbox}\, t \in \mathcal{T}[\![\mathrm{Str}(A) \to \mathrm{Str}(B)]\!]^{H^k(A)}_{\sharp}$, which in turn implies that $\mathrm{unbox}\, t \in \mathcal{T}[\![\mathrm{Str}(A) \to \mathrm{Str}(B)]\!]^{H^k(A)}_{\sharp l^* \mapsto v :: l^* \checkmark l^* \mapsto v' :: l^*}$. Therefore, we have that $\mathrm{unbox}\, t\, (\mathrm{adv}\, l^*) \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^k(A)}_{\sharp l^* \mapsto v :: l^* \checkmark l^* \mapsto v' :: l^*}$.

(ii) Let $\langle t, \eta \rangle \in T_{k+1}(A,B)$ and $\vdash v : A$. We need to find l, η_N, η_L, and $\vdash v' : B$ such that

$$\langle t; \sharp\eta, l^* \mapsto v :: l^* \checkmark l^* \mapsto \langle\rangle \rangle \Downarrow \langle v' :: l; \sharp\eta_N \checkmark \eta_L, l^* \mapsto \langle\rangle \rangle \tag{1}$$

$$\text{and} \quad \mathrm{adv}\, l \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^k(A)}_{\sharp\eta_L, l^* \mapsto w :: l^* \checkmark l^* \mapsto w' :: l^*} \quad \text{for all } w, w' \in \mathcal{V}[\![A]\!]^{()}_{\bot}. \tag{2}$$

By Lemma 6.5 (i), $v \in \mathcal{V}[\![A]\!]^{()}_{\bot}$, and by Lemma 6.5 (ii), there is some $w^* \in \mathcal{V}[\![A]\!]^{()}_{\bot}$. Since $t \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^{k+1}(A)}_{\sharp\eta, l^* \mapsto v :: l^* \checkmark l^* \mapsto w^* :: l^*}$, we have that

$$\langle t; \sharp\eta, l^* \mapsto v :: l^* \checkmark l^* \mapsto w^* :: l^* \rangle \Downarrow \langle v''; \sigma \rangle \text{ and } v'' \in \mathcal{V}[\![\mathrm{Str}(B)]\!]^{H^{k+1}(A)}_{\sigma}$$

Consequently, $v'' = v' :: l$ for some $v' \in \mathcal{V}[\![B]\!]^{H^{k+1}(A)}_{\sigma}$ by Lemma 6.1, σ is of the form $\sharp\eta_N \checkmark \eta_L, l^* \mapsto w^* :: l^*$. By Lemma 6.7 and Lemma 6.5 (i), we then have (1) and $\vdash v' : B$, respectively.

Finally, to prove (2), we assume $w, w' \in \mathcal{V}[\![A]\!]^{()}_{\bot}$ and show $\mathrm{adv}\, l \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^k(A)}_{\sharp\eta_L, l^* \mapsto w :: l^* \checkmark l^* \mapsto w' :: l^*}$. Since $t \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^{k+1}(A)}_{\sharp\eta, l^* \mapsto v :: l^* \checkmark l^* \mapsto w :: l^*}$, we have that

$$\langle t; \sharp\eta, l^* \mapsto v :: l^* \checkmark l^* \mapsto w :: l^* \rangle \Downarrow \langle v'''; \sigma' \rangle \text{ and } v''' \in \mathcal{V}[\![\mathrm{Str}(B)]\!]^{H^{k+1}(A)}_{\sigma'}$$

By Lemma 6.7 and Proposition 6.4 we thus know that $v''' = v' :: l$ and $\sigma' = \sharp\eta_N \checkmark \eta_L, l^* \mapsto w :: l^*$. Consequently, $\sigma'(l) \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^k}_{\sharp\eta_L, l^* \mapsto w :: l^* \checkmark l^* \mapsto w' :: l^*}$, which implies that

$$\mathrm{adv}\, l \in \mathcal{T}[\![\mathrm{Str}(B)]\!]^{H^k(A)}_{\sharp\eta_L, l^* \mapsto w :: l^* \checkmark l^* \mapsto w' :: l^*}.$$

□

7 RELATED WORK

The central ideas of functional reactive programming were originally developed for the language Fran [Elliott and Hudak 1997] for reactive animation. These ideas have since been developed into general purpose libraries for reactive programming, most prominently the Yampa library [Nilsson et al. 2002] for Haskell, which has been used in a variety of applications including games, robotics, vision, GUIs, and sound synthesis. Some of these libraries use a continuous notion of time, allowing

e.g., integrals over input data to be computed. A continuous notion of time can be encoded in Simply RaTT as well given that the language is extended with a type Time that suitably represents positive time intervals (e.g., floating-point numbers). For example, a Yampa-style signal function type from A to B is thus encoded as $\Box(\mathsf{Str}(\mathsf{Time}) \to \mathsf{Str}(A) \to \mathsf{Str}(B))$. This encoding reflects the (unoptimised) definition of Yampa-style signal functions [Nilsson et al. 2002], which is a coinductive type satisfying $\mathsf{SF}\ A\ B \cong \mathsf{Time} \to A \to (B \times \mathsf{SF}\ A\ B)$. We believe that it should be possible to implement a Yampa-style FRP library in Simply RaTT, and Section 4 has some examples of combinators similar to those found in Yampa. While some of these combinators have stability constraints on types, we believe that these constraints will always be satisfied in concrete applications.

Simply RaTT follows a *pull*-based approach to FRP, which means that the program is performing computations at every time step even if no event occurred. Elliott [2009] proposed an implementation of an FRP library that combines *pull*-based FRP with a *push*-based approach, where computation is only performed in response to incoming events. Whereas a pull-based approach is appropriate for example in games, which run at a fixed sampling rate, a push-based approach is more efficient for applications like GUIs, which often only need to react to events that occur infrequently.

The idea of using modal type operators for reactive programming goes back at least to the independent works of Jeffrey [2012]; Krishnaswami and Benton [2011] and Jeltsch [2013]. One of the inspirations for Jeffrey [2012] was to use linear temporal logic [Pnueli 1977] as a programming language through the Curry-Howard isomorphism. The work of Jeffrey and Jeltsch has mostly been based on denotational semantics, and Cave et al. [2014]; Krishnaswami [2013]; Krishnaswami and Benton [2011]; Krishnaswami et al. [2012] are the only works to our knowledge giving operational guarantees. The work of Cave et al. [2014] shows how one can encode notions of fairness in modal FRP, if one replaces the guarded fixed point operator with more standard (co)recursion for (co)inductive types.

The guarded recursive types and fixed point combinator originate with Nakano [2000], but have since been used for constructing logics for reasoning about advanced programming languages [Birkedal et al. 2011] using an abstract form of step-indexing [Appel and McAllester 2001]. The Fitch-style approach to modal types [Fitch 1952] has been used for guarded recursion in Clocked Type Theory [Bahr et al. 2017], where contexts can contain multiple, named ticks. Ticks can be used for reasoning about guarded recursive programs. The denotational semantics of Clocked Type Theory [Mannaa and Møgelberg 2018] reveals the difference from the more standard two-context approaches to modal logics, such as Dual Intuitionistic Linear Logic [Barber 1996]: In the latter, the modal operator is implicitly applied to the type of all variables in one context, in the Fitch-style, placing a tick in a context corresponds to applying a *left adjoint* to the modal operator to the context. Guatto [2018] introduced the notion of time warp and the warping modality, generalising the delay modality in guarded recursion, to allow for a more direct style of programming for programs with complex input-output dependencies. Combining these ideas with the garbage collection results of this paper, however, seems very difficult.

The previous work closest to the present work is that of Krishnaswami [2013]. We have already compared to this several times above, but give a short summary here. Simply RaTT is expressive enough to encompass all the positive examples of Krishnaswami's calculus, but we go a step further and identify a source of time leaks which allows us to eliminate in typing a number of leaking examples typable in Krishnaswami's calculus including the *leakyNats* example from the introduction, and *scary_const*. One might claim that these are explicit leaks, but detecting them in the type system is a major step forward we believe. Note that the Fitch-style approach is a real shift in approach: The time dependencies have changed, and Krishnaswami's context of stable variables has been replaced by a context of initial variables. One difference between these is that variables can be introduced from Krishnaswami's stable context. In Simply RaTT, initial variables

can generally not be introduced into temporal judgements. We plan to explain this change in terms of denotational semantics in future work.

Another approach to reactive programming is that of synchronous dataflow languages. Here the main abstraction is that of a "logical tick" or synchronous abstraction. This is the assumption that at each tick, the output is computed instantaneously from the input. This abstraction makes reasoning about time much easier than if we had to consider both the reactive behaviour and the internal timing behaviour of a program. Of particular interest is the synchronous dataflow language Lustre [Caspi et al. 1987]. Lustre is a first-order language used for describing and verifying real-time systems and is at the core of the SCADE industrial environment [Esterel Technologies SA 2019a] which is used for critical control systems in aerospace, rail transportation, industrial systems and nuclear power plants [Esterel Technologies SA 2019b]. In Section 5, we have shown how to encode some of the simpler concepts of Lustre in Simply RaTT, and how the concept of a logical tick fits well with the notion of stepwise stream unfolding.

8 CONCLUSIONS AND FUTURE WORK

We have presented the modal calculus Simply RaTT for reactive programming. Using the Fitch-style approach to modal types this gives a significant simplification of the type system and programming examples over existing approaches, in particular the calculus of Krishnaswami [2013]. Moreover, we have identified a source of time leaks and designed the type system to rule these out.

In future work we aim to extend Simply RaTT to a full type theory with dependent types for expressing properties of programs. Before doing that, however, we would like to extend Simply RaTT to encode fairness in types as in the work of Cave et al. [2014]. This is not easy, since it requires a distinction between inductive and coinductive guarded types, but Nakano's fixed point combinator forces these to coincide.

ACKNOWLEDGMENTS

This work was supported by a research grant (13156) from VILLUM FONDEN.

REFERENCES

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (Sept. 2001), 657–683. https://doi.org/10.1145/504709.504712 00283.

Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. 2017. The clocks are ticking: No more delays!. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/LICS.2017.8005097

Andrew Barber. 1996. *Dual intuitionistic linear logic.* Technical Report. University of Edinburgh, Edinburgh, UK.

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2011. First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. In *In Proc. of LICS*. IEEE Computer Society, Washington, DC, USA, 55–64. https://doi.org/10.2168/LMCS-8(4:1)2012

Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John A. Plaice. 1987. LUSTRE: A Declarative Language for Real-time Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '87)*. ACM, New York, NY, USA, 178–188. https://doi.org/10.1145/41625.41641

Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. 2014. Fair Reactive Programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, San Diego, California, USA, 361–372. https://doi.org/10.1145/2535838.2535881

Ranald Clouston. 2018. Fitch-style modal lambda calculi. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.), Vol. 10803. Springer, Springer International Publishing, Cham, 258–275.

Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. 2018. Modal Dependent Type Theory and Dependent Right Adjoints. *CoRR* abs/1804.05236 (2018), 1–21. arXiv:1804.05236 http://arxiv.org/abs/1804.05236

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, New York, NY, USA, 263–273. https://doi.org/10.1145/258948.

258973

Conal M. Elliott. 2009. Push-pull Functional Reactive Programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1596638.1596643

Pascal Raymond Erwan Jahier and Nicolas Halbwachs. 2019. The LUSTRE V6 Reference Manual. https://www-verimag.imag.fr/DIST-TOOLS/SYNCHRONE/lustre-v6/doc/lv6-ref-man.pdf

Esterel Technologies SA. 2019a. Scientific Background. http://www.esterel-technologies.com/about-us/scientific-historic-background/.

Esterel Technologies SA. 2019b. Success Stories. http://www.esterel-technologies.com/success-stories/.

Frederic Benton Fitch. 1952. *Symbolic logic, an introduction*. Ronald Press Co., New York, NY, USA.

Adrien Guatto. 2018. A generalized modality for recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*. ACM, 482–491.

Alan Jeffrey. 2012. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*, Koen Claessen and Nikhil Swamy (Eds.). ACM, Philadelphia, PA, USA, 49–60. https://doi.org/10.1145/2103776.2103783

Alan Jeffrey. 2014. Functional Reactive Types. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (CSL-LICS '14)*. ACM, New York, NY, USA, Article 54, 9 pages. https://doi.org/10.1145/2603088.2603106

Wolfgang Jeltsch. 2013. Temporal Logic with "Until", Functional Reactive Programming with Processes, and Concrete Process Categories. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification (PLPV '13)*. ACM, New York, NY, USA, 69–78. https://doi.org/10.1145/2428116.2428128

Neelakantan R. Krishnaswami. 2013. Higher-order Functional Reactive Programming Without Spacetime Leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, Boston, Massachusetts, USA, 221–232. https://doi.org/10.1145/2500365.2500588

Neelakantan R. Krishnaswami and Nick Benton. 2011. Ultrametric Semantics of Reactive Programs. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE Computer Society, Washington, DC, USA, 257–266. https://doi.org/10.1109/LICS.2011.38

Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. 2012. Higher-order functional reactive programming in bounded space. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, Philadelphia, PA, USA, 45–58. https://doi.org/10.1145/2103656.2103665

Bassel Mannaa and Rasmus Ejlers Møgelberg. 2018. The Clocks They Are Adjunctions: Denotational Semantics for Clocked Type Theory. In *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK (LIPIcs)*, Hélène Kirchner (Ed.), Vol. 108. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, New York, NY, USA, 23:1–23:17. https://doi.org/10.4230/LIPIcs.FSCD.2018.23

Per Martin-Löf and Giovanni Sambin. 1984. *Intuitionistic type theory*. Vol. 9. Bibliopolis Naples, Napoli, IT.

Conor McBride and Ross Paterson. 2008. Applicative programming with effects. *Journal of functional programming* 18, 1 (2008), 1–13.

Hiroshi Nakano. 2000. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*. IEEE Computer Society, Washington, DC, USA, 255–266. https://doi.org/10.1109/LICS.2000.855774

Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 51–64. https://doi.org/10.1145/581690.581695

Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Washington, DC, USA, 46–57. https://doi.org/10.1109/SFCS.1977.32

Marc Pouzet. 2006. Lucid synchrone, version 3. *Tutorial and reference manual. Université Paris-Sud, LRI* 1 (2006), 25.

Chapter 3

A Note on Categorical Semantics for Simply RaTT

A Note on Categorical Semantics for Simply RaTT

Christian Graulund

1 Introduction

Functional reactive programming (FRP) [EH97] is the application of techniques from functional programming to the domain of reactive programming, i.e, programming with values that *change over time* called signals. In recent years, there has been a growing interest in using *modalities* in the type systems for FRP[Kri13, CFPP14, Jef12, Jef14, BGM19]. In these, one or more modalities are added to a base language, and used to express various properties of FRP. In particular, all of them contain a "later" or "next" modality, usually denoted $\bigcirc$. This modality expresses that a type in only available "later". Adding this allows for *reasoning about time* on the type level. Most of them also contain an "always" modality, usually denoted $\Box$, expressing that a type is available at all timesteps or is "stable". In addition, several of these systems use *guarded recursion* [Nak00] to ensure causality and productivity of recursive definition. A subset of these[BGM19, BGM20] use the Fitch-style approach to modal type systems [Fit52, Clo18]. While the categorical semantics of Fitch-style modal type systems has been studied [Clo18, CMM$^+$18, MM18], there has not been, as far as we know, any work in detailing categorical semantics for Fitch-style modal languages for FRP.

In the setting of FRP, a lot of consideration has to be given to the operational behavior of programs. In particular, FRP is susceptible to various form of *spaceleaks*. A spaceleak happens when a program accumulates date over time in a manner not intended by the programmer. Avoiding these requires strong garbage collection properties. This problem was studied by Krishnaswami [Kri13] who provided a novel heap based operational semantics and proved the absence of a large class of spaceleaks. In their work, Bahr et. al. [BGM19] used a similar operational semantics for a Fitch-style calculus, named Simply RaTT, and proved it free of so-called *implicit* spaceleaks.

1.1 Contributions

In this work, we describe categorical semantics for Simply RaTT. The base category, $\mathcal{R}$, is a presheaf category over a suitable category of worlds. The worlds mirror the worlds used in the logical relation given in Bahr et. al. [BGM19], and hence, has the structure necessary to ensure operational properties. We introduce an abstract notion of garbage collection through a garbage collection functor, GC. This functor is additionally an idempotent comonad. We show how values are interpreted as coalgebras over GC. Further, the fact that GC is idempotent implies that the counit in an isomorphism, and hence, for all values $A \cong \mathsf{GC}(A)$.

To give the interpretation of the later modality $\bigcirc$, we introduce a pair of adjoint functors, $\Downarrow \dashv \Uparrow$, which describe the abstract notion of a timestep. These functors are defined between $\mathcal{R}$ and the co-Eilenberg-Moore category of GC-coalgebras.

To give the interpretation of the box modality $\Box$, we introduce another pair of adjoint functors, $\sqcup \dashv \sqcap$, which describes "time-independent" elements of an object.

To give the interpretation of terms, we define a commutative reader-like monad, $\mathcal{T}(-)$.

All of the above have non-trivial interactions, and the interpretations depends on this interaction.

Finally, we define a map for the interpretation of the fix point operator which is defined using a step-indexing approach.

2 Syntax

In this section we present the syntax and typing rules for Simply RaTT.

2.1 Types, Terms and Values

Definition 2.1. The types, terms and values of Simply RaTT is given by the grammars:

$$\begin{aligned} A, B ::= {} & \alpha \mid \mathsf{1} \mid \mathsf{Nat} \mid A \times B \mid A + B \mid A \to B \mid \bigcirc A \mid \Box A \mid \mu\alpha.A \\ v, w ::= {} & \langle\rangle \mid \bar{n} \mid \lambda x.t \mid \langle v, w\rangle \mid \mathsf{in}_i\, v \mid \mathsf{box}\, t \mid \mathsf{into}\, v \mid \mathsf{fix}\, x.t \\ s, t ::= {} & x \mid t_1 t_2 \mid \langle s, t\rangle \mid \pi_i\, t \mid \mathsf{in}_i\, t \mid \mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2 \\ \mid {} & \mathsf{delay}\, t \mid \mathsf{adv}\, t \mid \mathsf{unbox}\, t \mid \mathsf{into}\, t \mid \mathsf{out}\, t \end{aligned}$$

We denote the set of type and terms by Type and Term, respectively

2.2 Tokens and Contexts

Definition 2.2. A *token* is one of either $\checkmark$ or $\sharp$. The first is pronounced "tick" and the second "lock".

Definition 2.3. Given a countable set of term variables, denoted $x, y, z, \ldots$, a well-formed context is generated by the following rules:

$$\frac{}{\emptyset \vdash} \qquad \frac{\Gamma \vdash}{\Gamma, x : A \vdash} \qquad \frac{\Gamma \vdash \quad \mathsf{token\text{-}free}(\Gamma)}{\Gamma, \sharp \vdash} \qquad \frac{\Gamma \vdash \quad \mathsf{tick\text{-}free}(\Gamma) \quad \sharp \in \Gamma}{\Gamma, \checkmark \vdash}$$

where $\mathsf{token\text{-}free}(\cdot)$ denotes that a context contains neither a lock or a tick and $\mathsf{tick\text{-}free}(\cdot)$ denotes that a context does not contain a tick, respectively.

2.3 Well-formed Types and Terms

Definition 2.4. Given a type variable context Θ, a well-formed type is generated by the following rules:

$$\frac{\alpha \in \Theta}{\Theta \vdash \alpha : \mathsf{type}} \qquad \frac{}{\Theta \vdash 1 : \mathsf{type}} \qquad \frac{}{\Theta \vdash \mathsf{Nat} : \mathsf{type}} \qquad \frac{\Theta \vdash A : \mathsf{type} \qquad \Theta \vdash B : \mathsf{type}}{\Theta \vdash A \times B : \mathsf{type}}$$

$$\frac{\Theta \vdash A : \mathsf{type} \qquad \Theta \vdash B : \mathsf{type}}{\Theta \vdash A + B : \mathsf{type}} \qquad \frac{\Theta \vdash A : \mathsf{type} \qquad \Theta \vdash B : \mathsf{type}}{\Theta \vdash A \to B : \mathsf{type}} \qquad \frac{\Theta \vdash A : \mathsf{type}}{\Theta \vdash \bigcirc A : \mathsf{type}}$$

$$\frac{\Theta \vdash A : \mathsf{type}}{\Theta \vdash \Box A : \mathsf{type}} \qquad \frac{\Theta, \alpha \vdash A : \mathsf{type}}{\Theta \vdash \mu\alpha.A : \mathsf{type}}$$

Definition 2.5. We defined a predicate on types denoted stable. This is generated by the following rules:

$$\frac{}{1\ \mathsf{stable}} \qquad \frac{}{\mathsf{Nat}\ \mathsf{stable}} \qquad \frac{}{\Box A\ \mathsf{stable}} \qquad \frac{A\ \mathsf{stable} \qquad B\ \mathsf{stable}}{A \times B\ \mathsf{stable}} \qquad \frac{A\ \mathsf{stable} \qquad B\ \mathsf{stable}}{A + B\ \mathsf{stable}}$$

Definition 2.6. Given a well-formed context Γ, a well-formed term is generated by the following rules:

1: Simply typed λ-calculus:

$$\frac{\Gamma, x : A, \Gamma' \vdash \qquad A\ \mathsf{stable}\ \text{or}\ \mathsf{token\text{-}free}(\Gamma')}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \langle\rangle : 1} \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash \bar{n} : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash s : \mathsf{Nat} \qquad \Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash s + t : \mathsf{Nat}} \qquad \frac{\Gamma, x : A \vdash t : B \qquad \mathsf{tick\text{-}free}(\Gamma)}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash t' : A}{\Gamma \vdash t\, t' : B}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash t' : B}{\Gamma \vdash \langle t, t' \rangle : A \times B} \qquad \frac{\Gamma \vdash t : A_1 \times A_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \pi_i\, t : A_i} \qquad \frac{\Gamma \vdash t : A_i \qquad i \in \{1, 2\}}{\Gamma \vdash \mathsf{in}_i\, t : A_1 + A_2}$$

$$\frac{\Gamma, x : A_i \vdash t_i : B \qquad \Gamma \vdash t : A_1 + A_2 \qquad i \in \{1, 2\}}{\Gamma \vdash \mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2 : B}$$

2: Reactive terms:

$$\frac{\Gamma, \checkmark \vdash t : A}{\Gamma \vdash \mathsf{delay}\, t : \bigcirc A} \qquad \frac{\Gamma \vdash t : \bigcirc A}{\Gamma, \checkmark, \Gamma' \vdash \mathsf{adv}\, t : A} \qquad \frac{\Gamma \vdash t : \Box A}{\Gamma, \sharp, \Gamma' \vdash \mathsf{unbox}\, t : A} \qquad \frac{\Gamma, \sharp \vdash t : A}{\Gamma \vdash \mathsf{box}\, t : \Box A}$$

3: Guarded recursive terms:

$$\frac{\Gamma \vdash t : A[\bigcirc(\mu\alpha.A)/\alpha]}{\Gamma \vdash \mathsf{into}\, t : \mu\alpha.A} \qquad \frac{\Gamma \vdash t : \mu\alpha.A}{\Gamma \vdash \mathsf{out}\, t : A[\bigcirc(\mu\alpha.A)/\alpha]} \qquad \frac{\Gamma, \sharp, x : \bigcirc A \vdash t : A}{\Gamma \vdash \mathsf{fix}\, x.t : \Box A}$$

3 Worlds

In this section we present the category of worlds. These consists of three parts: A store, a sequence of heaps and an ordinal. The first two are for delayed terms and future inputs, respectively, whereas the ordinal is used for step-indexing of guarded terms.

3.1 Heaps, Sequences and Stores

Definition 3.1. We assume a countable set of variables, which we call *locations*. We will denote the set Loc and use $l, l', l'', \ldots$ to denote a single locations.

Definition 3.2. A *heap* is a finite mapping $\mathsf{Loc} \to \mathsf{Type}$. We will use η to denote a single heap. We write the domain of η as $\mathsf{dom}\,(\eta)$ and we write a heap lookup as $\eta(l)$. We write $\emptyset$ for the empty mapping, and given a heap η, a location l s.t. $l \notin \mathsf{dom}\,(\eta)$ and a type A, we write $\eta, l \mapsto A$ for the extension of η with the mapping of l to A.

Definition 3.3. We define an preorder on heaps, denoted $\eta \leq^h \eta'$ given by the standard ordering on finite maps, i.e., given two heaps η, η' we have $\eta \leq^h \eta'$ if and only if

$$\forall l \in \mathsf{dom}\,(\eta)\,.\eta(l) = A \Rightarrow \eta'(l) = A$$

That is, η is smaller than η' if η' contains all labels defined in η and they agree on all those.

Definition 3.4. We define a *heap sequence* as an infinite sequence of heaps, denoted $\eta; \eta'; \ldots$.

Definition 3.5. We define a preorder on heaps sequences, written $\overline{\eta} \leq^H \overline{\eta}'$, as:

$$\frac{\forall i.\eta_i \leq^h \eta'_i}{\eta_1; \eta_2; \ldots \leq^H \eta'_1; \eta'_2; \ldots}$$

Definition 3.6. A *store*, denoted σ, is given by the following grammar:

$$\bullet \mid \eta \mid \eta \checkmark \eta'$$

where η and η' are heaps. We call $\bullet$ the *nullstore*. Given a store σ and a location l, we define a store lookup as the partial function

$$\sigma(l) = \begin{cases} \eta(l) & \sigma = \eta \\ \eta'(l) & \sigma = \eta \checkmark \eta' \end{cases}$$

Given a store σ, a fresh location l and a type A, we write $\sigma, l \mapsto A$ as a short-hand for the partial function

$$\sigma, l \mapsto A = \begin{cases} (\eta, l \mapsto A) & \sigma = \eta \\ \eta \checkmark (\eta', l \mapsto A) & \sigma = \eta \checkmark \eta' \end{cases}$$

Definition 3.7. We define two preorders on stores denoted $\sigma \leq^s \sigma'$ and $\sigma \leq^s_\checkmark \sigma'$. These are given by the following rules:

$$\frac{}{\bullet \leq^s \bullet} \qquad \frac{\eta \leq^h \eta'}{\eta \leq^s \eta'} \qquad \frac{\eta_1 \leq^h \eta'_1 \qquad \eta_2 \leq^h \eta'_2}{\eta_1 \checkmark \eta_2 \leq^s \eta'_1 \checkmark \eta'_2}$$

and

$$\frac{\sigma \leq^s \sigma'}{\sigma \leq^s_{\checkmark} \sigma'} \qquad\qquad \frac{\eta \leq^h \eta'}{\eta \leq^s_{\checkmark} \eta'' \checkmark \eta'}$$

Definition 3.8. We define *garbage collection* on a store σ as the function:

$$\mathsf{gc}(\sigma) := \begin{cases} \eta' & \text{if } \sigma = \eta \checkmark \eta' \\ \sigma & \text{otherwise} \end{cases}$$

Lemma 3.9. *Garbage collection is idempotent, i.e,* $\mathsf{gc}(\mathsf{gc}(\sigma)) = \mathsf{gc}(\sigma)$.

Proof. We proceed by cases on σ:

- $\sigma = \eta$:
  By definition of garbage collection we have $\mathsf{gc}(\eta) = \eta$, and thus also $\mathsf{gc}(\mathsf{gc}(\eta)) = \mathsf{gc}(\eta) = \eta$ as wanted.
- $\sigma = \eta \checkmark \eta'$:
  By definition of garbage collection we have $\mathsf{gc}(\eta \checkmark \eta') = \eta'$ and then $\mathsf{gc}(\mathsf{gc}(\eta \checkmark \eta')) = \mathsf{gc}(\eta') = \eta'$, hence $\mathsf{gc}(\mathsf{gc}(\sigma)) = \mathsf{gc}(\sigma)$ as wanted.

□

Lemma 3.10. *Given stores* σ, σ' *s.t.* $\sigma \leq^s_{\checkmark} \sigma'$ *then* $\mathsf{gc}(\sigma) \leq^s \mathsf{gc}(\sigma')$.

Proof. By cases on $\sigma \leq^s_{\checkmark} \sigma'$:

- $\dfrac{\sigma \leq^s \sigma'}{\sigma \leq^s_{\checkmark} \sigma'}$:
  We proceed by cases on $\sigma \leq^s \sigma'$:
  - $\dfrac{}{\bullet \leq^s \bullet}$:
    Since $\mathsf{gc}(\bullet) = \bullet$ the conclusion follows by immediately.
  - $\dfrac{\eta \leq^h \eta'}{\eta \leq^s \eta'}$:
    Since $\mathsf{gc}(\eta) = \eta$ and $\mathsf{gc}(\eta') = \eta'$ we need to show $\eta \leq^s \eta'$ but this follows since we have $\eta \leq^h \eta'$ by assumption.
  - $\dfrac{\eta_1 \leq^h \eta_1' \qquad \eta_2 \leq^h \eta_2'}{\eta_1 \checkmark \eta_2 \leq^s \eta_1' \checkmark \eta_2'}$:
    By definition we have $\mathsf{gc}(\eta_1 \checkmark \eta_2) = \eta_2$ and $\mathsf{gc}(\eta_1' \checkmark \eta_2') = \eta_2'$ and we thus have to show $\eta_2 \leq^s_{\checkmark} \eta_2'$. By assumption we have $\eta_2 \leq^h \eta_2'$ which implies $\eta_2 \leq^s \eta_2'$.
- $\dfrac{\eta \leq^h \eta'}{\eta \leq^s_{\checkmark} \eta'' \checkmark \eta'}$:
  By definition $\mathsf{gc}(\eta) = \eta$ and $\mathsf{gc}(\eta'' \checkmark \eta') = \eta'$. We thus need to show $\eta \leq^s_{\checkmark} \eta'$. By assumption we have $\eta \leq^h \eta'$ which implies $\eta \leq^s \eta'$.

□

Lemma 3.11. *Given any store σ, we have $\mathsf{gc}(\sigma) \leq^s_\checkmark \sigma$.*

Proof. We split into cases for σ. In the cases for $\sigma = \bullet$ and $\sigma = \eta$, the conclusion follows by reflexivity since $\mathsf{gc}(\sigma) = \sigma$. Assume now $\sigma = \eta_1 \checkmark \eta_2$. We need to show $\eta_2 \leq^s_\checkmark \eta_1 \checkmark \eta_2$, but this follows by definition of the ticked store ordering and reflexivity. □

Definition 3.12. We define a *world* denoted w, as a triple $\langle \sigma, \overline{\eta}, \alpha \rangle$ where σ is a store, $\overline{\eta}$ is a heap sequence and α is an ordinal s.t $\alpha \leq \omega$. We extend the notion of garbage collection to worlds and write $\mathsf{gc}(\langle \sigma, \overline{\eta}, \alpha \rangle)$ to mean $\langle \mathsf{gc}(\sigma), \overline{\eta}, \alpha \rangle$.

Definition 3.13. We define two preorders on worlds, denoted $w \leq w'$ and $w \leq_\checkmark w'$, as

$$\langle \sigma, \overline{\eta}, \alpha \rangle \leq_\checkmark \langle \sigma', \overline{\eta}', \alpha' \rangle \Leftrightarrow (\sigma \leq^s_\checkmark \sigma') \wedge (\overline{\eta} \leq^H \overline{\eta}') \wedge (\alpha' \leq \alpha)$$
$$\langle \sigma, \overline{\eta}, \alpha \rangle \leq \langle \sigma', \overline{\eta}', \alpha' \rangle \Leftrightarrow (\sigma \leq^s \sigma') \wedge (\overline{\eta} \leq^H \overline{\eta}') \wedge (\alpha' \leq \alpha)$$

where $\alpha' \leq \alpha$ is the usual ordering on ordinals. We call $w \leq_\checkmark w'$ the *ticked* preorder.

Definition 3.14. We define the partial *upshift* function on worlds as:

$$\uparrow \langle \sigma, (\eta; \overline{\eta}), \alpha + 1 \rangle := \langle \mathsf{gc}(\sigma) \checkmark \eta, \overline{\eta}, \alpha \rangle \qquad \sigma \neq \bullet$$

Note that $\uparrow$ is defined exactly when $\alpha > 0$ and $\sigma \neq \bullet$.

Lemma 3.15. *Given worlds w, w' s.t. $\uparrow w$ and $\uparrow w'$ are defined then*

$$w \leq_\checkmark w' \Rightarrow \uparrow w \leq_\checkmark \uparrow w'$$

Proof. Assuming $\uparrow w$ and $\uparrow w'$ to be defined amounts to assuming $w = \langle \sigma, (\eta; \overline{\eta}), \alpha + 1 \rangle$ and $w' = \langle \sigma', (\eta'; \overline{\eta}'), \alpha' + 1 \rangle$ s.t. $\sigma \neq \bullet$ and $\sigma' \neq \bullet$. By assumption we have $\sigma \leq^s_\checkmark \sigma'$ and then by Lemma 3.10 we get $\mathsf{gc}(\sigma) \leq^s_\checkmark \mathsf{gc}(\sigma')$ and hence we get directly $\langle \mathsf{gc}(\sigma) \checkmark \eta, \overline{\eta}, \alpha \rangle \leq_\checkmark \langle \mathsf{gc}(\sigma') \checkmark \eta', \overline{\eta}', \alpha' \rangle$ as wanted. □

Lemma 3.16. *Given worlds w, w' s.t. $w \leq_\checkmark w'$ and $\uparrow w'$ is defined, then so is $\uparrow w$.*

Proof. Assuming that $\uparrow w'$ is defined amounts to assuming $w' = \langle \sigma', \overline{\eta}', \alpha' \rangle$ s.t. $\sigma' \neq \bullet$ and $\alpha' > 0$. Let $w = \langle \sigma, \overline{\eta}, \alpha \rangle$. By definition we know $\sigma \leq^s_\checkmark \sigma'$ and $\alpha' \leq \alpha$ which implies that $\sigma' \neq \bullet$ and $\alpha > 0$, respectively. Hence, we see that $\uparrow w$ is defined. □

Corollary 3.17. *Given worlds w, w' s.t. $w \leq_\checkmark w'$ and $\uparrow w'$ is defined, then so is $\uparrow w$ and $\uparrow w \leq_\checkmark \uparrow w'$.*

Proof. Follows from Lemma 3.15 and Lemma 3.16. □

Lemma 3.18. *Given any world w, s.t. $\uparrow w$ is defined, then $\uparrow w = \uparrow(\mathsf{gc}(w))$.*

Proof. Assuming $\uparrow w$ to be defined, amounts to assuming $w = (\sigma, (\eta; \overline{\eta}), \alpha + 1)$ where $\sigma \neq \bullet$. We now see

$$\begin{aligned}
\uparrow w &= \uparrow(\sigma, (\eta; \overline{\eta}), \alpha + 1) \\
&= (\mathsf{gc}(\sigma) \checkmark \eta, \overline{\eta}, \alpha) \\
&\overset{3.9}{=} (\mathsf{gc}(\mathsf{gc}(\sigma)) \checkmark \eta, \overline{\eta}, \alpha) \\
&= \uparrow(\mathsf{gc}(\sigma), (\eta; \overline{\eta}), \alpha + 1) \\
&= \uparrow(\mathsf{gc}(w))
\end{aligned}$$

□

Definition 3.19. We define the partial *downshift* map on worlds as:

$$\downarrow\langle \eta \checkmark \eta', \overline{\eta}, \alpha\rangle := \langle \eta, (\eta'; \overline{\eta}), \alpha + 1\rangle$$

Note that $\downarrow$ is defined for $\langle \sigma, \overline{\eta}, \alpha\rangle$ exactly when $\sigma = \eta \checkmark \eta'$.

Lemma 3.20. *Given worlds w, w' s.t. $\downarrow w$ and $\downarrow w'$ are defined then*

$$w \leq_{\checkmark} w' \Rightarrow \downarrow w \leq_{\checkmark} \downarrow w'.$$

Proof. Assuming $\downarrow w$ and $\downarrow w'$ to be defined amounts to assuming $w = \langle \eta_1 \checkmark \eta_2, \overline{\eta}, \alpha\rangle$ and $w' = \langle \eta_1' \checkmark \eta_2', \overline{\eta}', \alpha'\rangle$. It then follows immediately that $\langle \eta_1, (\eta_2; \overline{\eta}), \alpha + 1\rangle \leq_{\checkmark} \langle \eta_1', (\eta_2'; \overline{\eta}'), \alpha' + 1\rangle$ as wanted. □

Lemma 3.21. *Given worlds w, w' s.t. $w \leq_{\checkmark} w'$ and $\downarrow w$ is defined, then so is $\downarrow w'$.*

Proof. Let $w' = \langle \sigma', \overline{\eta}', \alpha'\rangle$. Assuming that $\downarrow w$ is defined amounts to assuming $w = \langle \eta_1 \checkmark \eta_2, \overline{\eta}, \alpha\rangle$. It now follows immediately that $\downarrow w'$ is defined since w' must be of the form $\sigma = \langle \eta_1' \checkmark \eta_2', \overline{\eta}', \alpha'\rangle$. □

Corollary 3.22. *Given worlds w, w' s.t. $w \leq_{\checkmark} w'$ and $\downarrow w$ is defined, then so is $\downarrow w'$ and $\downarrow w \leq_{\checkmark} \downarrow w'$.*

Proof. Follows by Lemma 3.20 and Lemma 3.21. □

Lemma 3.23. *Given a world w s.t. $\downarrow w$ is defined then*

$$\mathsf{gc}(\downarrow w) = \downarrow w$$

Proof. This follows immediately by definition of $\downarrow w$ and $\mathsf{gc}(w)$. □

Lemma 3.24. *Given a world w s.t. $\downarrow w$ is defined, then $\uparrow(\downarrow w) = w$.*

Proof. Assuming that $\downarrow w$ is defined amounts to $w = \langle \eta \checkmark \eta', \overline{\eta}, \alpha\rangle$ where $\overline{\eta}$ and α can be any heap sequence and any ordinal. We then have

$$\uparrow(\downarrow\langle \eta \checkmark \eta', \overline{\eta}, \alpha\rangle) = \uparrow\langle \eta, (\eta'; \overline{\eta}), \alpha + 1\rangle = \langle \mathsf{gc}(\eta) \checkmark \eta', \overline{\eta}, \alpha\rangle = \langle \eta \checkmark \eta', \overline{\eta}, \alpha\rangle$$

□

Lemma 3.25. *Given a world w s.t. $\uparrow w$ is defined, then $\downarrow(\uparrow w) \leq_{\checkmark} w$.*

Proof. Assuming that $\uparrow w$ is defined amounts to assuming $w = \langle \sigma, (\eta'; \overline{\eta}), \alpha + 1\rangle$ where $(\eta'; \overline{\eta})$ and α can be any heap sequence and ordinal and $\sigma \neq \bullet$. We split into two cases for σ:

- $\sigma = \eta$:
  We then have

  $$\downarrow(\uparrow\langle \eta, (\eta'; \overline{\eta}), \alpha + 1\rangle) = \downarrow\langle \mathsf{gc}(\eta) \checkmark \eta', \overline{\eta}, \alpha\rangle = \downarrow\langle \eta \checkmark \eta', \overline{\eta}, \alpha\rangle = \langle \eta, (\eta'; \overline{\eta}), \alpha + 1\rangle$$

  Since $\downarrow(\uparrow w) = w$ it follows be reflexivity that $\downarrow(\uparrow w) \leq_{\checkmark} w$.

- $\sigma = \eta \checkmark \eta''$:
  We then have

$$\downarrow(\uparrow\langle \eta \checkmark \eta'', (\eta'; \overline{\eta}), \alpha + 1\rangle) = \downarrow\langle \mathsf{gc}(\eta \checkmark \eta'') \checkmark \eta', \overline{\eta}, \alpha\rangle = \downarrow\langle \eta'' \checkmark \eta', \overline{\eta}, \alpha\rangle = \langle \eta'', (\eta'; \overline{\eta}), \alpha + 1\rangle$$

  Note now that $\eta'' \leq^s_\checkmark \eta \checkmark \eta''$ and hence we have $\downarrow(\uparrow w) \leq_\checkmark w$ as wanted.

□

Lemma 3.26. *For any world $(\sigma, \overline{\eta}, \alpha)$ s.t $\sigma \neq \bullet$ and $\alpha = \alpha' + 1$ we have $\downarrow(\uparrow(\mathsf{gc}(w)) = \mathsf{gc}(w)$.*

Proof. Since $\sigma \neq \bullet$, we either have $\sigma = \eta' \checkmark \eta$ or $\sigma = \eta$. In either case $\mathsf{gc}(\sigma) = \eta$. We now see

$$\downarrow(\uparrow\langle \eta, (\eta'; \overline{\eta}), \alpha' + 1\rangle) = \downarrow\langle \mathsf{gc}(\eta) \checkmark \eta', \overline{\eta}, \alpha'\rangle = \downarrow\langle \eta \checkmark \eta', \overline{\eta}, \alpha'\rangle = \langle \eta, (\eta'; \overline{\eta}), \alpha' + 1\rangle$$

□

Lemma 3.27. *Let w, w' be worlds such that $\downarrow w$ and $\uparrow w'$ is defined. Then there is a bi-implication*

$$\downarrow w \leq_\checkmark w' \Leftrightarrow w \leq_\checkmark \uparrow w'$$

Proof. We show the implication in both direction

- Assume $\downarrow w \leq_\checkmark w'$. By Lemma 3.24 we know that $\uparrow(\downarrow w)$ is defined and $\uparrow(\downarrow w) = w$. By Lemma 3.15 we then have $w \leq_\checkmark \uparrow w'$ as wanted.
- Assume $w \leq_\checkmark \uparrow w'$. By Lemma 3.20 have $\downarrow w \leq_\checkmark \downarrow\uparrow w'$ and then by Lemma 3.25 and transitivity we have $\downarrow w \leq_\checkmark w'$ as wanted.

□

Definition 3.28. Given a world $w = (\sigma, \overline{\eta}, \alpha)$ we define projection maps for each of the components, and denote these as $w.\sigma, w.\overline{\eta}$ and $w.\alpha$, respectively.

4 Categorical Structure

In this section, we define the categories $\mathcal{R}$ and $\mathcal{R}^{\mathsf{GC}}$ together with two pairs of adjoint functors, namely $\Downarrow \dashv \Uparrow$ and $\sqcup \dashv \sqcap$ which are needed to interpret $\bigcirc$ and $\Box$, respectively.

4.1 The Category $\mathcal{R}$

Definition 4.1. The category $\mathcal{R}$ is the category of covariant presheaves over the ticked preorder on worlds. Given $w \leq_\checkmark w'$ and $a \in A(w)$, we write $a_{w \leq_\checkmark w'}$ for the application of the induced map $A(w) \to A(w')$ or just $a_{w'}$ when the context is clear. We will sometimes work with the morphism directly, and then we will denote it $(-)|_{w'} : A(w) \to A(w')$.

The initial, terminal and natural number object of $\mathcal{R}$ are given by the constant functor over the initial, terminal and natural number object of Set, respectively. We denote these by $1_\mathcal{R}, \emptyset_\mathcal{R}$ and $\mathbb{N}_\mathcal{R}$.

Being a presheaf category, $\mathcal{R}$ is both complete and cocomplete and hence, has all limits and colimits.

Definition 4.2. Given that $\mathcal{R}$ is a presheaf category, it is cartesian closed. Given two objects A, B in $\mathcal{R}$ and some world w, the internal hom, denoted B^A, is given at w by

$$B^A(w) := \left\{ (f_{w'})_{w \leq_\checkmark w'} \;\middle|\; f_{w'} : A(w') \to B(w') \text{ s.t. } \begin{array}{ccc} A(w) & \xrightarrow{f_w} & B(w) \\ \downarrow{\scriptstyle w \leq_\checkmark w'} & & \downarrow{\scriptstyle w \leq_\checkmark w'} \\ A(w') & \xrightarrow{f_{w'}} & B(w') \end{array} \right\}$$

with the evaluation map $\mathsf{ev} : B^A \times A \to B$ being the natural transformation that is given at w by

$$\mathsf{ev}(w) = \lambda \langle f, a \rangle . f_w(a)$$

i.e., for $w \leq_\checkmark w'$, the following commutes

$$\begin{array}{ccc} (B^A \times A)(w) & \xrightarrow{\mathsf{ev}_w} & B(w) \\ \downarrow{\scriptstyle w \leq_\checkmark w'} & & \downarrow{\scriptstyle w \leq_\checkmark w'} \\ (B^A \times A)(w') & \xrightarrow{\mathsf{ev}_{w'}} & B(w') \end{array}$$

Definition 4.3. We define the morphism $\mathsf{plus} : \mathbb{N}_\mathcal{R} \times \mathbb{N}_\mathcal{R} \to \mathbb{N}_\mathcal{R}$ as pointwise addition.

4.2 The Category $\mathcal{R}^{\mathsf{GC}}$ of Garbage Collection Coalgebras

Definition 4.4. We define *garbage collection*, denoted $\mathsf{GC}(\cdot)$, on $\mathcal{R}$ as the endofunctor:

$$\mathsf{GC}(A)(\sigma, \overline{\eta}, \alpha) = A(\mathsf{gc}(\sigma), \overline{\eta}, \alpha)$$
$$\mathsf{GC}(f)(\sigma, \overline{\eta}, \alpha) = f(\mathsf{gc}(\sigma), \overline{\eta}, \alpha)$$

Remark 4.5. By definition of GC and the pointwise structure of $\mathcal{R}$, it follows that $\mathsf{GC}(A \times B) = \mathsf{GC}(A) \times \mathsf{GC}(B)$. To see this, consider a world w. We then have

$$\begin{aligned} \mathsf{GC}(A \times B)(w) &= (A \times B)(\mathsf{gc}(w)) \\ &= A(\mathsf{gc}(w)) \times B(\mathsf{gc}(w)) \\ &= \mathsf{GC}(A)(w) \times \mathsf{GC}(B)(w) \\ &= (\mathsf{GC}(A) \times \mathsf{GC}(B))(w) \end{aligned}$$

We will use this fact throughout the following.

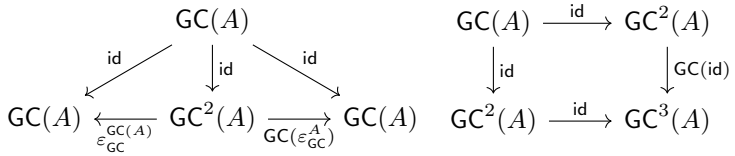
Lemma 4.6. *The garbage collection endofunctor is idempotent, i.e, for any object A, $\mathsf{GC}^2(A) = \mathsf{GC}(A)$*

Proof. This follows directly from Lemma 3.9. □

Definition 4.7. Given a presheaf A, there is a morphism $\mathsf{GC}(A) \to A$ induced by Lemma 3.11, namely $a \mapsto a_{(\mathsf{gc}(\sigma) \leq_\checkmark \sigma)}$. For a given object A, we denote this map $\varepsilon^A_{\mathsf{GC}}$ or just $\varepsilon_{\mathsf{GC}}$ when the context is clear.

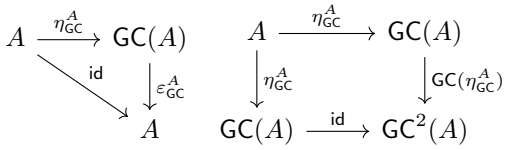
Lemma 4.8. *The garbage collection endofunctor carries an idempotent comonad structure, with counit given by $\varepsilon_{\mathsf{GC}}$ defined in Definition 4.7 and the comultiplication given by the identity.*

Proof. That the comonad is idempotent follows immediately by Lemma 4.6. To see why the above induces a comonad we need to show that the following two diagrams commute



which follows easily. Note both that $\mathsf{GC}(\varepsilon^A_{\mathsf{GC}})$ and $\varepsilon^{\mathsf{GC}(A)}_{\mathsf{GC}}$ are the identity. □

Definition 4.9. We denote the co-Eilenberg-Moore category of GC-coalgebras over the comonad GC by $\mathcal{R}^{\mathsf{GC}}$. The objects of $\mathcal{R}^{\mathsf{GC}}$ are pairs $(A, \eta^A_{\mathsf{GC}})$ where A is an object of $\mathcal{R}$ and η^A_{GC} is a morphism $A \to \mathsf{GC}(A)$ s.t. the following diagrams commutes



Lemma 4.10. *Given any* GC*-coalgebra* $(A, \eta^A_{\mathsf{GC}})$*, then* η^A_{GC} *is an isomorphism.*

Proof. This follows by Proposition 4.3.2 in The Handbook of Categorical Algebra volume 2 [Bor94] and the fact that the maps $\mathsf{GC}(\varepsilon^A_{\mathsf{GC}}) : \mathsf{GC}^2(A) \to \mathsf{GC}(A)$ and $\varepsilon^{\mathsf{GC}(A)}_{\mathsf{GC}} : \mathsf{GC}^2(A) \to \mathsf{GC}(A)$ are both the identity and hence equal. □

Remark 4.11. From the above lemma we see that a GC-coalgebra is precisely an object of $\mathcal{R}$ where $\varepsilon_{\mathsf{GC}}$ is an isomorphism.

4.3 The $\Downarrow \dashv \Uparrow$ Adjunction

Definition 4.12. We define the *upshift* functor $\Uparrow(-) : \mathcal{R} \to \mathcal{R}^{\mathsf{GC}}$.

$$(\Uparrow A)(w) = \begin{cases} A(\uparrow w) & \uparrow w \text{ defined} \\ 1 & \text{otherwise} \end{cases} \qquad (\Uparrow f)(w) = \begin{cases} f(\uparrow w) & \uparrow w \text{ defined} \\ ! & \text{otherwise} \end{cases}$$

First, we show that if A is an object of $\mathcal{R}$, then $\Uparrow A$ in an object of $\mathcal{R}$. Assume worlds w, w' s.t. $w \leq_{\checkmark} w'$. We want to construct a morphism $(\Uparrow A)(w) \to (\Uparrow A)(w')$. We have two cases depending on whether $\uparrow w'$ is defined. If $\uparrow w'$ is not defined, then $(\Uparrow A)(w') = 1$ and the map is the unique map into the terminal object. If $\uparrow w'$ is defined, then by Corollary 3.17, $\uparrow w$ is also defined and we know $\uparrow w \leq_{\checkmark} \uparrow w'$ which, since A is an object of $\mathcal{R}$, induces a map $A(\uparrow w) \to A(\uparrow w')$ as wanted.

We now proceed to show that if A is an object of $\mathcal{R}$, then $\Uparrow A$ is a GC-coalgebra. Assume a world w. We have two cases depending on whether $\uparrow w$ is defined. If $\uparrow w$ is not defined, then $(\Uparrow A)(w) = 1$ and thus we trivially have $\mathsf{GC}((\Uparrow A))(w) = 1 = (\Uparrow A)(w)$. If $\uparrow w$ is defined, we have

$$(\mathsf{GC}(\Uparrow A))(w) = (\Uparrow A)(\mathsf{gc}(w)) = A(\uparrow(\mathsf{gc}(w))) \overset{3.18}{=} A(\uparrow w) = (\Uparrow A)(w).$$

It follows easily that $\Uparrow(f)$ has the correct source and target, and respects identities and composition.

Definition 4.13. We define the *downshift* functor $\Downarrow(-) : \mathcal{R}^{\mathsf{GC}} \to \mathcal{R}$.

$$(\Downarrow A)(w) = \begin{cases} A(\downarrow w) & \downarrow w \text{ defined} \\ \emptyset & \text{otherwise} \end{cases} \qquad (\Downarrow f)(w) = \begin{cases} f(\downarrow w) & \downarrow w \text{defined} \\ ! & \text{otherwise} \end{cases}$$

We first show that if A is an object of $\mathcal{R}^{\mathsf{GC}}$, then $\Downarrow A$ is an object of $\mathcal{R}$: Assume worlds w, w' s.t. $w \leq_{\checkmark} w'$. We want to construct a morphism $(\Downarrow A)(w) \to (\Downarrow A)(w')$. We have two cases depending on whether $\downarrow w$ is defined. If $\downarrow w$ is not defined, then $(\Downarrow A)(w) = \emptyset$ and the map is the unique map out of the initial object. If $\downarrow w$ is defined, then by Corollary 3.22, $\downarrow w'$ is also defined and we know $\downarrow w \leq_{\checkmark} \downarrow w'$ which, since A is an object of $\mathcal{R}$, induces a map $A(\downarrow w) \to A(\downarrow w')$ as wanted.

It follows easily that $\Downarrow(f)$ has the correct source and target, and respects identities and composition.

Theorem 4.14. The down- and upshift endofunctors form an adjunction:

$$\Downarrow \dashv \Uparrow$$

Proof. To prove the adjunction we show that there is a natural isomorphism of homsets:

$$\mathsf{Hom}_{\mathcal{R}}(\Downarrow A, B) \cong \mathsf{Hom}_{\mathcal{R}^{\mathsf{GC}}}(A, \Uparrow B)$$

We define maps in both directions

- $\tau_{\Downarrow\Uparrow} : \mathsf{Hom}_{\mathcal{R}}(\Downarrow A, B) \to \mathsf{Hom}_{\mathcal{R}^{\mathsf{GC}}}(A, \Uparrow B)$:
  Let $f \in \mathsf{Hom}_{\mathcal{R}}(\Downarrow A, B)$, we define

  $$(\tau_{\Downarrow\Uparrow}(f))(w) = \begin{cases} f(\uparrow(\mathsf{gc}(w))) & \uparrow w \text{ defined} \\ ! & \text{otherwise} \end{cases}$$

  To see why this is well-defined, note that if $\uparrow w$ is not defined, then $(\Uparrow B)(w) = 1$. If $\uparrow w$ is defined, $f(\uparrow(\mathsf{gc}(w))$ is a map $(\Downarrow A)(\uparrow(\mathsf{gc}(w)) \to B(\uparrow(\mathsf{gc}(w)))$ and we see that

  $$(\Downarrow A)(\uparrow(\mathsf{gc}(w))) = A(\downarrow(\uparrow(\mathsf{gc}(w)))) = A(\mathsf{gc}(w)) = \mathsf{GC}(A)(w) \cong A(w)$$

  where the second equality is by Lemma 3.26 and the last isomorphism is by Lemma 4.10. Further we have

  $$B(\uparrow(\mathsf{gc}(w))) = B(\uparrow(w)) = (\Uparrow B)(w)$$

  where the first equality is by Lemma 3.18. We thus have $\tau_{\Downarrow\Uparrow}(f) \in \mathsf{Hom}_{\mathcal{R}^{\mathsf{GC}}}(A, \Uparrow B)$ as wanted.

- $\tau^{-1}_{\Downarrow\Uparrow} : \mathsf{Hom}_{\mathcal{R}^{\mathsf{GC}}}(A, \Uparrow B) \to \mathsf{Hom}_{\mathcal{R}}(\Downarrow A, B)$:
  Let $g \in \mathsf{Hom}_{\mathcal{R}^{\mathsf{GC}}}(A, \Uparrow B)$, we define

  $$(\tau^{-1}_{\Downarrow\Uparrow}(g))(w) \begin{cases} g(\downarrow w) & \downarrow w \text{ defined} \\ ! & \text{otherwise} \end{cases}$$

To see why this is well-defined, note that if $\downarrow w$ is not defined, then $(\Downarrow A)(w) = \emptyset$. If $\downarrow w$ is defined, $g(\downarrow w)$ is a morphism $A(\downarrow w) \to (\Uparrow B)(\downarrow w)$ and we see

$$(\Uparrow B)(\downarrow w) = B(\uparrow\downarrow(w)) = B(w)$$

where the second equality is by Lemma 3.24. By definition we have $A(\downarrow w) = (\Downarrow A)(w)$. We thus have $\tau^{-1}_{\Downarrow\Uparrow}(g) \in \mathsf{Hom}_{\mathcal{R}}(\Downarrow A, B)$ as wanted.

We now need to show that the above morphisms are mutually inverse.

- $(\tau^{-1}_{\Downarrow\Uparrow} \circ \tau_{\Downarrow\Uparrow})(f) = f$:
  Given $f \in \mathsf{Hom}_{\mathcal{R}}(\Downarrow A, B)$ and some world w, we split into cases. If $\downarrow w$ is defined, then we have
  $$((\tau^{-1}_{\Downarrow\Uparrow} \circ \tau_{\Downarrow\Uparrow})(f))(w) = (\tau_{\Downarrow\Uparrow}(f))(\downarrow w) = f(\uparrow(\mathsf{gc}(\downarrow w))) = f(w)$$
  where the last equation follows by Lemma 3.18 and Lemma 3.24. If $\downarrow w$ is not defined, note that $(\Downarrow A)(w) = \emptyset$ and hence $f(w) = !$. We see
  $$((\tau^{-1}_{\Downarrow\Uparrow} \circ \tau_{\Downarrow\Uparrow})(f))(w) = ! = f(w)$$
  as wanted.

- $(\tau_{\Downarrow\Uparrow} \circ \tau^{-1}_{\Downarrow\Uparrow})(g) = g$:
  Given $g \in \mathsf{Hom}_{\mathcal{R}^{\mathsf{GC}}}(A, \Uparrow B)$ and some world w, we split into cases. if $\uparrow w$ is defined, then
  $$(\tau_{\Downarrow\Uparrow} \circ \tau^{-1}_{\Downarrow\Uparrow}(g))(w) = (\tau^{-1}_{\Downarrow\Uparrow}(g))(\uparrow(\mathsf{gc}(w))) = g(\downarrow(\uparrow(\mathsf{gc}(w)))) = g(\mathsf{gc}(w)) = g(w)$$
  where the third equality is by Lemma 3.26 and the fourth follows by the fact that g is a morphism of GC-coalgebras. If $\uparrow w$ is not defined, then $(\Uparrow B)(w) = \mathsf{1}$ and hence $g(w) = !$. We see
  $$((\tau_{\Downarrow\Uparrow} \circ \tau^{-1}_{\Downarrow\Uparrow})(g))(w) = ! = g(w)$$
  as wanted.

□

4.4 The $\sqcup \dashv \sqcap$ Adjunction

Definition 4.15. We define the endofunctor $\sqcap(-) : \mathcal{R} \to \mathcal{R}$

$$\sqcap(A)(w) = \begin{cases} \lim\limits_{\overline{\eta}} A(\emptyset, \eta, w.\alpha) & w.\sigma = \bullet \\ \mathsf{1} & w.\sigma \neq \bullet \end{cases} \qquad \sqcap(f)(w) = \begin{cases} \lambda w'.\lambda a.\langle f_{(\emptyset,\overline{\eta},w'.\alpha)}((\pi_{\overline{\eta}})(a))\rangle_{\overline{\eta}} & w.\sigma = \bullet \\ ! & w.\sigma \neq \bullet \end{cases}$$

where $\langle - \rangle_{\overline{\eta}}$ denotes the paring into the limit, and $\pi_{\overline{\eta}}(-)$ denotes the projection out of the limit.

To see why this is well-defined, note that for $w.\sigma \neq \bullet$ it is trivially well-defined. Now, to see why $\sqcap A$ is an object of $\mathcal{R}$, assume worlds $w \leq_{\checkmark} w'$ and we need to construct a map $(\sqcap A)(w) \to (\sqcap A)(w')$. If $w.\sigma = \bullet$, then, by definition, also $w'.\sigma = \bullet$ and hence this should be a map $\lim\limits_{\overline{\eta}} A(\emptyset, \overline{\eta}, w.\alpha) \to \lim\limits_{\overline{\eta}} A(\emptyset, \overline{\eta}, w'.\alpha)$. To give this map, it is sufficient to give it into $A(\emptyset, \overline{\eta}, w'.\alpha)$

for each $\overline{\eta}$, since we can then pair up these. For each $\overline{\eta}$, this map is the projection composed with weakening in A, i.e., for $a \in \varprojlim_{\overline{\eta}} A(\emptyset, \overline{\eta}, w.\alpha)$

$$a_{w'} = \langle \pi_{\overline{\eta}}(a)_{(\emptyset,\overline{\eta},w'.\alpha)} \rangle_{\overline{\eta}}$$

Assume now $f : A \to B$, to see that $(\sqcap f)$ has the correct source and target, note that given w s.t. $w.\sigma = \bullet$, this should be a map $(\sqcap A)(w) \to (\sqcap B)(w)$ s.t. it is natural in w. To that end, assume w' s.t. $w \leq_{\checkmark} w'$. Note that by definition $w'.\sigma = \bullet$ and hence this should be a map

$$\varprojlim_{\overline{\eta}} A(\emptyset, \overline{\eta}, w'.\alpha) \to \varprojlim_{\overline{\eta}} B(\emptyset, \overline{\eta}, w'.\alpha)$$

To give such a map, it is sufficient to give map into $B(\emptyset, \overline{\eta}, w'.\alpha)$ for all $\overline{\eta}$, since we can then pair up these. Assuming $\overline{\eta}$ and $a \in \varprojlim_{\overline{\eta}} A(\emptyset, \overline{\eta}, w'.\alpha)$, we see that $\pi_{\overline{\eta}}(a) : A(\emptyset, \overline{\eta}, w'.\alpha)$ and then

$$f_{(\emptyset,\overline{\eta},w'.\alpha)}(\pi_{\overline{\eta}}(a)) : B(\emptyset, \overline{\eta}, w'.\alpha)$$

as wanted. The naturality follows by naturality of f.

Definition 4.16. We define the endofunctor $\sqcup(-) : \mathcal{R} \to \mathcal{R}$

$$\sqcup(A)(w) = \begin{cases} \operatorname{colim}_{\overline{\eta}} A(\bullet, \overline{\eta}, w.\alpha) & w.\sigma \neq \bullet \\ \emptyset & w.\sigma = \bullet \end{cases} \qquad \sqcup(f)(w) = \begin{cases} \lambda w'.[\lambda a.\mathsf{in}_{\overline{\eta}}(f_{(\bullet,\overline{\eta},w'.\alpha)}(a))]_{\overline{\eta}} & w'.\sigma \neq \bullet \\ ! & w'.\sigma = \bullet \end{cases}$$

where $[-]_{\overline{\eta}}$ denotes the coparing out of the colimit and $\mathsf{in}_{\overline{\eta}}(-)$ denotes the injection into the colimit.

To see why this is well-defined, note that for $w.\sigma = \bullet$, this is trivially well-defined. Now, to see why $\sqcup A$ is an object of $\mathcal{R}$, assume worlds $w \leq_{\checkmark} w'$ and we need to construct a map $(\sqcup A)(w) \to (\sqcup A)(w')$. If $w.\sigma \neq \bullet$ then, by definition, $w'.\sigma \neq \bullet$ and hence this is a map $\operatorname{colim}_{\overline{\eta}} A(\bullet, \overline{\eta}, w.\alpha) \to \operatorname{colim}_{\overline{\eta}} A(\bullet, \overline{\eta}.w'.\alpha)$. To give this map, it is sufficient to give it out of $A(\bullet, \overline{\eta}, w.\alpha)$ for each $\overline{\eta}$, since we can then do the coparing of these. For each $\overline{\eta}$ this map is weakening composed with the injection, i.e, for $a \in \operatorname{colim}_{\overline{\eta}} A(\bullet, \overline{\eta}, w.\alpha)$

$$a_{w'} = [\lambda a'.\mathsf{in}_{\overline{\eta}}(a'_{(\bullet,\overline{\eta},w'.\alpha)})]_{\overline{\eta}}$$

Assume now $f : A \to B$, to see that $\sqcup f$ has the correct source and target, note that given w s.t. $w.\sigma \neq \bullet$, this should be a map $(\sqcup A)(w) \to (\sqcup B)(w)$ s.t. it is natural in w. To that end, assume w' s.t. $w \leq_{\checkmark} w'$. Note, by definition, $w'.\sigma \neq \bullet$ and hence this should be a map

$$\operatorname{colim}_{\overline{\eta}} A(\bullet, \overline{\eta}, w'.\alpha) \to \operatorname{colim}_{\overline{\eta}} B(\bullet, \overline{\eta}, w'.\alpha)$$

To give such a map, it is sufficient to give a map out of $A(\bullet, \overline{\eta}, w'.\alpha)$ for each $\overline{\eta}$, since we can then do the coparing of all of these. Assuming $\overline{\eta}$ and $a \in A(\bullet, \overline{\eta}, w'.\alpha)$, we see that $f_{(\bullet,\overline{\eta},w'.\alpha)}(a) : B(\bullet, \overline{\eta}, w'.\alpha)$ and then further $\mathsf{in}_{\overline{\eta}}(f_{(\bullet,\overline{\eta},w'.\alpha)}(a))$ as wanted. The naturality follows by naturality of f.

Lemma 4.17. *The $\sqcup$ and $\sqcap$ endofunctors form an adjunction:*

$$\sqcup \dashv \sqcap$$

Proof. We prove that there is a natural isomorphism of hom-sets

$$\mathsf{Hom}_{\mathcal{R}}(\sqcup A, B) \cong \mathsf{Hom}_{\mathcal{R}}(A, \sqcap B)$$

We define maps in both directions:

- $\tau_{\sqcup\sqcap} : \mathsf{Hom}_{\mathcal{R}}(\sqcup A, B) \to \mathsf{Hom}_{\mathcal{R}}(A, \sqcap B)$:
  Let $f \in \mathsf{Hom}_{\mathcal{R}}(\sqcup A, B)$, we define

  $$(\tau_{\sqcup\sqcap}(f))(\sigma, \overline{\eta}, \alpha) = \begin{cases} \langle f(\emptyset, \overline{\eta}', \alpha)\rangle_{\overline{\eta}'} \circ \mathsf{in}_{\overline{\eta}} & \sigma = \bullet \\ ! & \sigma \neq \bullet \end{cases}$$

  To see why this is well-defined, note first that in the case for $\sigma \neq \bullet$, then $(\sqcap B)(\sigma, \overline{\eta}, \alpha) = 1$. In the case for $\sigma = \bullet$, note first that given $A(\bullet, \overline{\eta}, \alpha)$, we have

  $$\mathsf{in}_{\overline{\eta}} : A(\bullet, \overline{\eta}, \alpha) \to \operatorname*{colim}_{\overline{\eta}'} A(\bullet, \overline{\eta}', \alpha)$$

  Now, given any $\overline{\eta}'$, we have a map

  $$f(\emptyset, \overline{\eta}', \alpha) : \operatorname*{colim}_{\overline{\eta}'} A(\bullet, \overline{\eta}', \alpha) \to B(\emptyset, \overline{\eta}', \alpha)$$

  and hence, by paring up all of these, we get a map

  $$\langle f(\emptyset, \overline{\eta}', \alpha)\rangle_{\overline{\eta}'} : \operatorname*{colim}_{\overline{\eta}'} \ A(\bullet, \overline{\eta}', \alpha) \to \lim_{\overline{\eta}'} B(\emptyset, \overline{\eta}', \alpha)$$

  and since $(\sqcap B)(\bullet, \overline{\eta}, \alpha) = \lim_{\overline{\eta}'} B(\emptyset, \overline{\eta}', \alpha)$ this has the correct type.

- $\tau^{-1}_{\sqcup\sqcap} : \mathsf{Hom}_{\mathcal{R}}(A, \sqcap B) \to \mathsf{Hom}_{\mathcal{R}}(\sqcup A, B)$:
  Let $g \in \mathsf{Hom}_{\mathcal{R}}(A, \sqcap B)$, we define

  $$((\tau_{\sqcup\sqcap})^{-1}(g))(\sigma, \overline{\eta}, \alpha) = \begin{cases} (-)|_{(\sigma,\overline{\eta},\alpha)} \circ \pi_{\overline{\eta}} \circ [g(\bullet, \overline{\eta}', \alpha)]_{\overline{\eta}'} & \sigma \neq \bullet \\ ! & \sigma = \bullet \end{cases}$$

  To see why this is well-defined, note first that in the case for $\sigma = \bullet$, $(\sqcup A)(\bullet, \overline{\eta}, \alpha) = \emptyset$. In the case for $\sigma \neq \bullet$, note that for any $\overline{\eta}'$, we have a map $g(\bullet, \overline{\eta}', \alpha) : A(\bullet, \overline{\eta}', \alpha) \to \lim_{\overline{\eta}''} B(\emptyset, \overline{\eta}'', \alpha)$ and hence, by doing the coparing of all of these, we get a map

  $$[g(\bullet, \overline{\eta}', \alpha)]_{\overline{\eta}'} : \operatorname*{colim}_{\overline{\eta}'} A(\bullet, \overline{\eta}', \alpha) \to \lim_{\overline{\eta}''} B(\emptyset, \overline{\eta}'', \alpha)$$

  Now, the projection out of the limit is the map $\pi_{\overline{\eta}} : \lim_{\overline{\eta}''} B(\emptyset, \overline{\eta}'', \alpha) \to B(\emptyset, \overline{\eta}, \alpha)$. Finally, we can apply the induced map $(-)_{(\sigma,\overline{\eta},\alpha)} : B(\emptyset, \overline{\eta}, \alpha) \to B(\sigma, \overline{\eta}, \alpha)$ and hence this has the correct type.

We now show that these are actually mutually inverse

- $((\tau^{-1}_{\sqcup\sqcap} \circ \tau_{\sqcup\sqcap})(f)) = f$
  Given $f \in \mathsf{Hom}_R(\sqcup A, B)$ and some world $(\sigma, \overline{\eta}, \alpha)$ we split into cases depending on whether $\sigma = \bullet$. If $\sigma = \bullet$, then $(\sqcup A)(\bullet, \overline{\eta}, \alpha) = \emptyset$ and hence $f(\bullet, \overline{\eta}, \alpha) = !$. We see

  $$((\tau^{-1}_{\sqcup\sqcap} \circ \tau_{\sqcup\sqcap})(f))(\bullet, \overline{\eta}, \alpha) = ! = f(\bullet, \overline{\eta}, \alpha)$$

  If $\sigma \neq \bullet$, we then have

  $$\begin{aligned}
  ((\tau^{-1}_{\sqcup\sqcap} \circ \tau_{\sqcup\sqcap})(f))(\sigma, \overline{\eta}, \alpha) &= \tau^{-1}_{\sqcup\sqcap}(\tau_{\sqcup\sqcap}(f))(\sigma, \overline{\eta}, \alpha) \\
  &= (-)|_{(\sigma,\overline{\eta},\alpha)} \circ \pi_{\overline{\eta}} \circ [\tau_{\sqcup\sqcap}(f)(\bullet, \overline{\eta}', \alpha)]_{\overline{\eta}'} \\
  &= (-)|_{(\sigma,\overline{\eta},\alpha)} \circ \pi_{\overline{\eta}} \circ [\langle f(\emptyset, \overline{\eta}'', \alpha)\rangle_{\overline{\eta}''} \circ \mathsf{in}_{\overline{\eta}'}]_{\overline{\eta}'} \\
  &= (-)|_{(\sigma,\overline{\eta},\alpha)} \circ \pi_{\overline{\eta}} \circ \langle f(\emptyset, \overline{\eta}'', \alpha)\rangle_{\overline{\eta}''} \\
  &= (-)|_{(\sigma,\overline{\eta},\alpha)} \circ f(\emptyset, \overline{\eta}, \alpha) \\
  &= f(\sigma, \overline{\eta}, \alpha)
  \end{aligned}$$

  The first three equalities are just writing out the definitions. The fourth equality follows from the naturality condition on the colimit, and is essentially the "eta" reduction rule for the colimit, i.e., doing a coparing of functions precomposed with the injection is the identity. The fifth equality follows from naturality of the limit and is essentially the "beta" reduction rule for the limit. Finally, the sixth equality is the naturality of morphisms.

- $((\tau_{\sqcup\sqcap} \circ \tau^{-1}_{\sqcup\sqcap})(f)) = f$:
  Given $g \in \mathsf{Hom}_R(A, \sqcap B)$ and some world $(\sigma, \overline{\eta}, \alpha)$ we split into cases depending on whether $\sigma = \bullet$. If $\sigma \neq \bullet$, then $(\sqcap B)(\sigma, \overline{\eta}, \alpha) = 1$ and hence $f(\sigma, \overline{\eta}, \alpha) = !$. We see

  $$((\tau_{\sqcup\sqcap} \circ \tau^{-1}_{\sqcup\sqcap})(f))(\sigma, \overline{\eta}, \alpha) = ! = f(\sigma, \overline{\eta}, \alpha)$$

  If $\sigma = \bullet$, we then have

  $$\begin{aligned}
  ((\tau_{\sqcup\sqcap} \circ \tau^{-1}_{\sqcup\sqcap})(f))(\bullet, \overline{\eta}, \alpha) &= \tau_{\sqcup\sqcap}(\tau^{-1}_{\sqcup\sqcap}(f))(\bullet, \overline{\eta}, \alpha) \\
  &= \langle \tau^{-1}_{\sqcup\sqcap}(f)(\emptyset, \overline{\eta}', \alpha)\rangle_{\overline{\eta}'} \circ \mathsf{in}_{\overline{\eta}} \\
  &= \langle (-)|_{(\emptyset,\overline{\eta}',\alpha)} \circ \pi_{\overline{\eta}'} \circ [f(\bullet, \overline{\eta}'', \alpha)]_{\overline{\eta}''}\rangle_{\overline{\eta}'} \circ \mathsf{in}_{\overline{\eta}} \\
  &= \langle \pi_{\overline{\eta}'} \circ [f(\bullet, \overline{\eta}'', \alpha)]_{\overline{\eta}''}\rangle_{\overline{\eta}'} \circ \mathsf{in}_{\overline{\eta}} \\
  &= [f(\bullet, \overline{\eta}'', \alpha)]_{\overline{\eta}''} \circ \mathsf{in}_{\overline{\eta}} \\
  &= f(\bullet, \overline{\eta}, \alpha)
  \end{aligned}$$

  The first three equalites are just writing out the definitions. The fourth equality follows since $(-)|_{(\emptyset,\overline{\eta}',\alpha)}$ is the map induced by $(\emptyset, \overline{\eta}', \alpha) \leq_{\checkmark} (\emptyset, \overline{\eta}', \alpha)$ which is just the identity. The fifth equality follows form the naturality condition on the limit and is the "eta" reduction rule for the limit, i.e., doing a pairing of functions postcomposed with the projection is the identity. The sixth equality follows from naturality of the colimit.

□

5 Categorical Semantics

In this section, we first given an interpretation of the types and contexts of Simply RaTT. We then define additional structure on these and finally, give an interpretation of well-typed terms.

5.1 Values, Terms, Stores and Contexts

Definition 5.1. We define the size of a type A, denote $|A|$ as:

$$\begin{aligned} |\alpha| = |\bigcirc A| = |\mathsf{Nat}| = |1| &= 1 \\ |A \times B| = |A + B| = |A \to B| &= 1 + |A| + |B| \\ |\Box A| = |\mu\alpha.A| &= 1 + |A| \end{aligned}$$

Definition 5.2. We define the store, value and term functors by mutual recursion. In particular, the following are defined by well-founded induction on the lexicographical ordering on the triple $(\alpha, |A|, e)$ where α is the ordinal component of the world, $|A|$ is the size of types defined above and $e = 1$ for the term interpretation and $e = 0$ for the value interpretation. Note that for this to make sense, we should inline the definition of the store interpretation into the term interpretation, s.t. we only define the value and term interpretation, but for readability, we keep the below presentation. To see why the below is indeed well-founded, note that in the interpretation of $\mu\alpha.A$, $A[\bigcirc\mu\alpha.A/\alpha]$ is strictly smaller then $\mu\alpha.A$ since $|\bigcirc\mu\alpha.A| = |\alpha|$. Note further that in the interpretation of $\bigcirc A$, the step-index is decreased by definition of $\Uparrow$.

$$\begin{aligned} &\mathcal{S}(-,-) : \mathcal{W}^{\mathsf{op}} \times \mathcal{W} \to \mathsf{Set} \\ &\mathcal{S}(w, w') := \begin{cases} \prod_{A \in \mathsf{codom}(\eta)} [\![A]\!](w') & w = (\eta \checkmark \eta', \overline{\eta}, \alpha) \\ 1 & \text{otherwise} \end{cases} \end{aligned}$$

$$\begin{aligned} [\![-]\!] &: \mathsf{Types} \to \mathcal{R} \\ [\![1]\!] &:= 1_{\mathcal{R}} \\ [\![\mathsf{Nat}]\!] &:= \mathbb{N}_{\mathcal{R}} \\ [\![A \times B]\!] &:= [\![A]\!] \times [\![B]\!] \\ [\![A + B]\!] &:= [\![A]\!] + [\![B]\!] \\ [\![A \to B]\!] &:= \mathsf{GC}(\mathcal{T}([\![B]\!])^{[\![A]\!]}) \\ [\![\bigcirc A]\!] &:= \Uparrow(\mathcal{T}([\![A]\!])) \\ [\![\Box A]\!] &:= \sqcap(\mathcal{T}([\![A]\!])) \\ [\![\mu\alpha.A]\!] &:= [\![A[\bigcirc(\mu\alpha.A)/\alpha]]\!] \end{aligned}$$

$$\begin{aligned} \mathcal{T}(-) &: \mathcal{R} \to \mathcal{R} \\ \mathcal{T}(A)(w) &:= \{ t \mid \forall w' \geq_{\checkmark} w.t(w') : \mathcal{S}(w', w') \to A(w') \} \end{aligned}$$

Given $w \in \mathcal{W}^{\mathsf{op}}$, $\mathcal{S}(w, -)$ is an object in $\mathcal{R}^{\mathsf{GC}}$, i.e., a presheaf and GC-coalgebra. This follows immediately by the fact that for any w', $\mathcal{S}(w, w')$ is a product of objects in $\mathcal{R}^{\mathsf{GC}}$. On the other hand, if we fix $w \in \mathcal{W}$, then given $w' \leq_{\checkmark} w''$, there is a projection map $\mathcal{S}(w'', w) \to \mathcal{S}(w', w)$ which simply removes all the types that are present in w'' but not in w'.

To see why $\mathcal{T}(A)$ is indeed a presheaf, consider $w \leq_{\checkmark} w'$. We then need to give a morphism $t_{w' \leq_{\checkmark} w} : \mathcal{T}(A)(w) \to \mathcal{T}(A)(w')$. To that end, assume $w' \leq_{\checkmark} w''$. Since $w \leq_{\checkmark} w''$ we see that $t(w'') : \mathcal{S}(w'', w'') \to A(w'')$ as wanted and hence, we define $t_{w' \leq_{\checkmark} w} = \lambda w''.t(w'')$. Given $f : A \to B$,

the action on morphism $\mathcal{T}(f) : \mathcal{T}(A) \to \mathcal{T}(B)$ is given by

$$\mathcal{T}(f)(w)(t) = \lambda w'.f(w') \circ t(w')$$

To see that this is natural, we need to show that given $w \leq_\checkmark w'$, the following diagram commutes

$$\begin{array}{ccc} \mathcal{T}(A)(w) & \xrightarrow{\mathcal{T}(f)(w)} & \mathcal{T}(B)(w) \\ {\scriptstyle w \leq_\checkmark w'}\downarrow & & \downarrow{\scriptstyle w \leq_\checkmark w'} \\ \mathcal{T}(A)(w') & \xrightarrow{\mathcal{T}(f)(w')} & \mathcal{T}(B)(w') \end{array}$$

Going from top left, going down and right gives

$$t \mapsto \lambda w''.t(w'') \mapsto \lambda w''.f(w'') \circ t(w'')$$

and going right and then down gives

$$t \mapsto \lambda w'.f(w') \circ t(w') \mapsto \lambda w''.f(w'') \circ t(w'')$$

and hence the diagram commutes.

Definition 5.3. We define the interpretation of a syntactic context into $\mathcal{R}$:

$$\begin{aligned} \mathcal{C}[\![-]\!] &: \mathsf{Context} \to \mathcal{R} \\ \mathcal{C}[\![\cdot]\!] &:= \lambda w. \begin{cases} 1 & w = (\bullet, \overline{\eta}, \alpha) \\ \emptyset & \text{otherwise} \end{cases} \\ \mathcal{C}[\![\Gamma, x : A]\!] &:= \mathcal{C}[\![\Gamma]\!] \times [\![A]\!] \\ \mathcal{C}[\![\Gamma, \checkmark]\!] &:= \Downarrow(\mathcal{C}[\![\Gamma]\!]) \\ \mathcal{C}[\![\Gamma, \sharp]\!] &:= \sqcup(\mathcal{C}[\![\Gamma]\!]) \end{aligned}$$

5.2 Monadic Structure of Term Interpretation

Lemma 5.4. *The term functor carries a monad structure where the unit and multiplication morphisms are given by*

$$\begin{aligned} \eta^A_{\mathcal{T}}(w) &: A(w) \to \mathcal{T}(A)(w) \\ \eta^A_{\mathcal{T}}(w) &= \lambda a.\lambda w'.\lambda s.a_{w'} \end{aligned}$$

$$\begin{aligned} \mu^A_{\mathcal{T}}(w) &: \mathcal{T}^2(A)(w) \to \mathcal{T}(A)(w) \\ \mu^A_{\mathcal{T}}(w) &= \lambda t.\lambda w'.\lambda s.t(w')(s)(w')(s) \end{aligned}$$

Proof. To see why the multiplication types, note that $t(w') : \mathcal{S}(w', w') \to \mathcal{T}(A)(w')$ and hence $t(w')(s) : \mathcal{T}(A)(w')$ and thus $t(w')(s)(w') : \mathcal{S}(w', w') \to A(w')$. First we show that these are actually morphisms in the category. For the unit we need to show that given $w \leq_\checkmark w'$, the following diagram commutes:

$$\begin{array}{ccc} A(w) & \xrightarrow{\eta^A_{\mathcal{T}}(w)} & \mathcal{T}(A)(w) \\ {\scriptstyle w \leq_\checkmark w'}\downarrow & & \downarrow{\scriptstyle w \leq_\checkmark w'} \\ A(w') & \xrightarrow{\eta^A_{\mathcal{T}}(w')} & \mathcal{T}(A)(w') \end{array}$$

Starting at the top left corner and going down then right we get

$$a \mapsto a_{w'} \mapsto \lambda w''.\lambda s.(a_{w'})_{w''}$$

Note that $(a_{w'})_{w''} = a_{w''}$. Going right and then down we get

$$a \mapsto \lambda w'.\lambda s.a_{w'} \mapsto \lambda w''.\lambda s.a_{w''}$$

and hence the diagram commutes. For the multiplication, given $w \leq_\checkmark w'$, we need to show the following diagram commutes

$$\begin{array}{ccc} \mathcal{T}^2(A)(w) & \xrightarrow{\mu^A_{\mathcal{T}}(w)} & \mathcal{T}(A)(w) \\ {\scriptstyle w \leq_\checkmark w'}\downarrow & & \downarrow{\scriptstyle w \leq_\checkmark w'} \\ \mathcal{T}^2(A)(w') & \xrightarrow{\mu^A_{\mathcal{T}}(w')} & \mathcal{T}(A)(w') \end{array}$$

Starting in the top left corner and going down and right, we have for $t \in \mathcal{T}^2(A)(w)$:

$$t \mapsto \lambda w''.t(w'') \mapsto \lambda w''.\lambda s.((t(w'')(s))(w''))(s)$$

If we instead go right and then down we have:

$$t \mapsto \lambda w'.\lambda s.(t(w')(s))(w')(s) \mapsto \lambda w''.\lambda s.(t(w'')(s))(w'')(s)$$

and hence the diagram commutes.

We now show that these are natural transformations. For the unit we need to show that the following diagram commutes, where $f : A \to B$:

$$\begin{array}{ccc} A(w) & \xrightarrow{\eta^A_{\mathcal{T}}(w)} & \mathcal{T}(A)(w) \\ {\scriptstyle f(w)}\downarrow & & \downarrow{\scriptstyle \mathcal{T}(f)(w)} \\ B(w) & \xrightarrow{\eta^B_{\mathcal{T}}(w)} & \mathcal{T}(B)(w) \end{array}$$

Starting in the top left corner and going down and then right, we have for $a \in A(w)$:

$$a \mapsto f(w)(a) \mapsto \lambda w'.\lambda s.(f(w)(a))_{w'}$$

If we instead go right and then down we have

$$a \mapsto \lambda w'.\lambda s.a_{w'} \mapsto \lambda w'.\lambda s.f(w')(a_{w'})$$

and since f is itself a natural transformation, we have $(f(w)(a))_{w'} = f(w')(a_{w'})$ as wanted.

For the multiplication we need to show the following diagram commutes, where $f : A \to B$:

$$\begin{array}{ccc} \mathcal{T}^2(A)(w) & \xrightarrow{\mu_{\mathcal{T}}^A(w)} & \mathcal{T}(A)(w) \\ {\scriptstyle \mathcal{T}(\mathcal{T}(f))(w)}\downarrow & & \downarrow{\scriptstyle \mathcal{T}(f)(w)} \\ \mathcal{T}^2(B)(w) & \xrightarrow{\mu_{\mathcal{T}}^B(w)} & \mathcal{T}(B)(w) \end{array}$$

Starting in the top left corner and going down and then right, we have for $t \in \mathcal{T}^2(A)(w)$:

$$\begin{aligned} t &\mapsto \lambda w'.\lambda s.\mathcal{T}(f)(w')(t(w')(s)) \\ &\mapsto \lambda w'.\lambda s.((\mathcal{T}(f)(w')(t(w')(s))))(w')(s) \\ &= \lambda w'.\lambda s.f(w')((t(w')(s))(w')(s)) \end{aligned}$$

If we instead go right and then down we have

$$\begin{aligned} t &\mapsto \lambda w'.\lambda s.(t(w')(s))(w')(s) \\ &\mapsto \lambda w'.\lambda s.f(w')((t(w')(s))(w')(s)) \end{aligned}$$

and hence the diagram commutes . □

Lemma 5.5. *The term monad is a strong monad with strength given by*

$$\begin{aligned} &\mathsf{str}_{A,B} : A \times \mathcal{T}(B) \to \mathcal{T}(A \times B) \\ &\mathsf{str}_{A,B} = \lambda w.\lambda \left\langle a, t \right\rangle .\lambda w'.\lambda s.\left\langle a_{w'}, t(w')(s) \right\rangle \end{aligned}$$

Proof. We show that the following diagrams commute:

- $$\left(\begin{array}{ccc} & \mathcal{T}(A) & \\ {\scriptstyle \cong}\swarrow & & \searrow{\scriptstyle \cong} \\ 1_{\mathcal{R}} \times \mathcal{T}(A) & \xrightarrow{\mathsf{str}_{1_{\mathcal{R}},A}} & \mathcal{T}(1_{\mathcal{R}} \times A) \end{array} \right):$$

  Given some world w we see

  $$t \cong \langle \bullet, t \rangle \mapsto \lambda w'.\lambda s.\left\langle \bullet, t(w')(s) \right\rangle \cong \lambda w'.\lambda s.t(w')(s) = t$$

- $$\left(\begin{array}{ccc} (A \times B) \times \mathcal{T}(C) & \xrightarrow{\mathsf{str}_{A\times B,C}} & \mathcal{T}((A \times B) \times C) \\ {\scriptstyle \cong}\downarrow & & \downarrow{\scriptstyle \cong} \\ A \times (B \times \mathcal{T}(C)) & \xrightarrow{\mathsf{id}_A \times \mathsf{str}_{B,C}} A \times \mathcal{T}(B \times C) & \xrightarrow{\mathsf{str}_{A,B\times C}} \mathcal{T}(A \times (B \times C)) \end{array} \right):$$

  Given some world w, if we start at the top left corner and go down and then right we get:

  $$\begin{aligned} \left\langle \left\langle a, b \right\rangle , t \right\rangle &\cong \left\langle a, \left\langle b, t \right\rangle \right\rangle \\ &\mapsto \left\langle a, \lambda w'.\lambda s.\left\langle b_{w'}, t(w')(s) \right\rangle \right\rangle \\ &\mapsto \lambda w'.\lambda s.\left\langle a_{w'}, \left\langle b_{w'}, t(w')(s) \right\rangle \right\rangle \end{aligned}$$

  If we instead go right and then down we get:

  $$\begin{aligned} \left\langle \left\langle a, b \right\rangle , tc \right\rangle &\mapsto \lambda w'.\lambda s.\left\langle \left\langle a, b \right\rangle_{w'} , t(w')(s) \right\rangle \\ &\cong \lambda w'.\lambda s.\left\langle a_{w'}, \left\langle b_{w'}, t(w')(s) \right\rangle \right\rangle \end{aligned}$$

  and hence the diagram commutes. Note that we use the fact that $\left\langle a, b \right\rangle_{w'} = \left\langle a_{w'}, b_{w'} \right\rangle$.

- $$\left(\begin{array}{ccc} & A \times B & \\ {\scriptstyle \mathsf{id}A \times \eta^B_{\mathcal{T}}} \swarrow & & \searrow {\scriptstyle \eta^{A\times B}_{\mathcal{T}}} \\ A \times \mathcal{T}(B) & \xrightarrow{\quad \mathsf{str}_{A,B} \quad} & \mathcal{T}(A \times B) \end{array} \right):$$

  Given some w and starting at the top, we see

  $$\begin{aligned} \langle a, b\rangle &\mapsto \langle a, \lambda w'.\lambda s.b_{w'}\rangle \\ &\mapsto \lambda w'.\lambda s.\, \langle a_{w'}, b_{w'}\rangle \\ &= \lambda w'.\lambda w.\, \langle a, b\rangle_{w'} \end{aligned}$$

  which is exactly $\eta^{A\times B}_{\mathcal{T}}(\langle a, b\rangle)$ as wanted.

- $$\left(\begin{array}{ccc} A \times \mathcal{T}(\mathcal{T}(B)) & \xrightarrow{\mathsf{str}_{A,\mathcal{T}(B)}} & \mathcal{T}(A \times \mathcal{T}(B)) \xrightarrow{\mathcal{T}(\mathsf{str}A,B)} \mathcal{T}(\mathcal{T}(A \times B)) \\ {\scriptstyle \mathsf{id}_A \times \mu^B_{\mathcal{T}}}\downarrow & & \downarrow {\scriptstyle \mu^{A\times B}_{\mathcal{T}}} \\ A \times \mathcal{T}(B) & \xrightarrow{\quad \mathsf{str}A,B \quad} & \mathcal{T}(A \times B) \end{array} \right):$$

  Given some world w and staring in the top left corner and going first down we see

  $$\begin{aligned} \langle a, ttb\rangle &\mapsto \langle a, \lambda w'.\lambda s.t((w')(s))(w')(s)\rangle \\ &\mapsto \lambda w'.\lambda s.\, \langle a_{w'}, t((w')(s))(w')(s)\rangle \end{aligned}$$

  If we instead go first right we see

  $$\begin{aligned} \langle a, ttb\rangle &\mapsto \lambda w'.\lambda s.\, \langle a_{w'}, t(w')(s)\rangle \\ &\mapsto (\lambda \langle a, t'\rangle .\lambda w''.\lambda s'.\, \langle a_{w''}, t'(w'')(s')\rangle) \circ (\lambda w'.\lambda s.\, \langle a_{w'}, t(w')(s)\rangle) \\ &= \lambda w''.\lambda s'.\lambda w'.\lambda s\, \langle (a_{w'})_{w''}, t(w')(s)(w'')(s')\rangle \\ &\mapsto \lambda w'.\lambda s.\, \langle a_{w'}, t(w')(s)(w')(s)\rangle \end{aligned}$$

  and hence the diagram commutes.

□

Lemma 5.6. *The term monad is a costrong monad with costrength given by*

$$\begin{aligned} &\mathsf{cstr}_{A,B} : \mathcal{T}(A) \times B \to \mathcal{T}(A \times B) \\ &\mathsf{cstr}_{A,B} = \lambda w.\lambda \langle ta, b\rangle .\lambda w'.\lambda s.\, \langle ta(w')(s), b_{w'}\rangle \end{aligned}$$

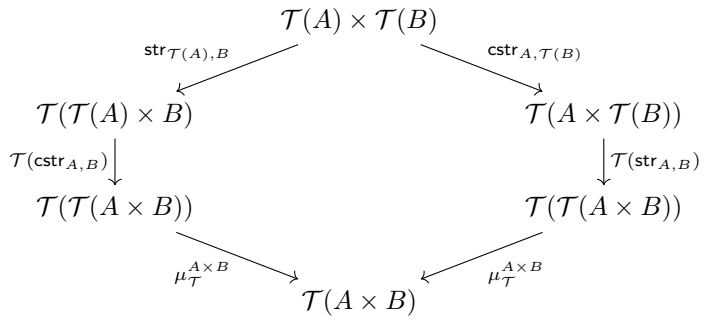
Proof. The proof follows by the exact same reasoning as in Lemma 5.5. □

Lemma 5.7. *The term monad is a commutative monad. We will write*

$$\mathsf{comm}_{A,B} : \mathcal{T}(A) \times \mathcal{T}(B) \to \mathcal{T}(A \times B)$$

to denote the map.

Proof. We need to show that the following diagram commutes:



Given some world w and starting at the top and following the left path we get

$$\begin{aligned}\langle ta, tb\rangle &\mapsto \lambda w'.\lambda s.\, \langle ta_{w'}, tb(w')(s)\rangle \\ &\mapsto \lambda\, \langle ta', b\rangle\, .\lambda w''\lambda s'.\, \langle ta'(w'')(s'), b_{w''}\rangle \circ \lambda w'.\lambda s.\, \langle ta_{w'}, tb(w')(s)\rangle \\ &= \lambda w''.\lambda s'.\lambda w'.\lambda s.\, \langle ta_{w'}(w'')(s''), (tb(w')(s))_{w''}\rangle \\ &\mapsto \lambda w'.\lambda s.\, \langle ta(w')(s), tb(w')(s)\rangle\end{aligned}$$

where we have used the fact that $ta_{w'}(w') = ta(w') = ta(w')_{w'}$. If we instead follow the right path we get

$$\begin{aligned}\langle ta, tb\rangle &\mapsto \lambda w'.\lambda s.\, \langle ta(w')(s), tb_{w'}\rangle \\ &\mapsto \lambda\, \langle a, tb'\rangle\, .\lambda w''.\lambda s'.\, \langle a_{w''}, tb'(w'')(s')\rangle \circ \lambda w'.\lambda s.\, \langle ta(w')(s), tb_{w'}\rangle \\ &= \lambda w''.\lambda s'.\lambda w'.\lambda s.\, \langle (ta(w')(s))_{w''}, tb_{w'}(w'')(s')\rangle \\ &\mapsto \lambda w'.\lambda s.\, \langle ta(w')(s), tb(w')(s)\rangle\end{aligned}$$

and hence the diagram commutes. □

5.3 Additional Structure for Interpretation

Lemma 5.8. *Given a type A, then $[\![A]\!]$ is a garbage collection coalgebra, i.e.,*

$$[\![A]\!] \cong \mathsf{GC}([\![A]\!])$$

Proof. We proceed by induction over the type A. Since $\mathbf{1}_{\mathcal{R}}$ and $\mathbb{N}_{\mathcal{R}}$ are constant presheaves, it follows immediately that they are also garbage collection coalgebras. The conclusion follows by the induction hypothesis for $A \times B$, $A + B$ and $\mu\alpha.A$. For $[\![\Box A]\!]$ it follows immediately by definition of $\sqcap$. Finally, that $[\![A \to B]\!]$ is a garbage collection coalgebras follows since GC is idempotent. That $[\![-]\!]$ actually produces an object of $\mathcal{R}^{\mathsf{GC}}$ follows immediately for $\mathbf{1}_{\mathcal{R}}, \mathbb{N}_{\mathcal{R}}$, $[\![A \times B]\!]$ and $[\![A + B]\!]$. For $[\![\bigcirc A]\!]$, it follows by definition of $\Uparrow$ and the definition of the term interpretation. For $[\![\Box A]\!]$, it follows immediately by definition of $\sqcap$ that it is a GC-coalgebra. Finally, that $[\![A \to B]\!]$ is an object of $\mathcal{R}^{\mathsf{GC}}$ follows immediately from the fact that GC is idempotent. □

Definition 5.9. We define the morphism $\mathsf{step} : \Downarrow\mathcal{T}(A) \to \Downarrow A$.

$$\mathsf{step} := \lambda w. \begin{cases} \mathsf{id}_{\mathbf{1}_{\mathcal{R}}}(w) & \downarrow w \text{ undefined} \\ \lambda f.f(\downarrow w) & \downarrow w \text{ defined} \end{cases}$$

To see why this is well-defined, assume some world w. If $\downarrow w$ is not defined, note that $(\Downarrow \mathcal{T}(A))(w) = (\Downarrow A)(w) = 1$. If $\downarrow w$ is defined we have

$$(\Downarrow \mathcal{T}(A))(w) = \{ f \mid \forall w' \geq_{\checkmark} \downarrow w . f(w') : \mathcal{S}(w', w') \to A(w') \}$$

and thus, for $f \in (\Downarrow \mathcal{T}(A))(w)$, we have

$$f(\downarrow w) : \mathcal{S}(\downarrow w, \downarrow w) \to A(\downarrow w).$$

Note that since $\downarrow w$ is assumed to be well-defined, it follows by definition that $\mathcal{S}(\downarrow w, \downarrow w) = 1_{\mathcal{R}}$, and hence $f(\downarrow w) : A(\downarrow w)$ as wanted.

Lemma 5.10. *Given $A \in \mathcal{R}$, we have the following equality*

$$\Downarrow(\mathsf{GC}(A)) = \Downarrow A$$

Proof. Assume some world w. If $\downarrow w$ is not defined, both sides are the terminal object and we are done. Assume now that $\downarrow w$ is defined. We then see

$$\Downarrow(\mathsf{GC}(A))(w) = \mathsf{GC}(A)(\downarrow w) = A(\mathsf{gc}(\downarrow w)) = A(\downarrow w) = (\Downarrow A)(w)$$

where the the third equality is by Lemma 3.23 and the rest are by definition. □

Lemma 5.11. *Given a syntactic context Γ s.t. Γ is tick-free, then the interpretation is a GC-coalgebra, i.e.,*

$$\mathcal{C}[\![\Gamma]\!] \cong \mathsf{GC}(\mathcal{C}[\![\Gamma]\!])$$

Proof. In the case that the interpretation is empty, the conclusion follows immediately. We now assume that the interpretation is non-empty and we proceed by induction on Γ. In the case that $\Gamma = \cdot$, then $\mathcal{C}[\![\Gamma]\!](\cdot, \overline{\eta}, \alpha) = \{\bullet\}$ and $\mathsf{GC}(\{\bullet\}) = \{\bullet\}$ and hence $\mathcal{C}[\![\Gamma]\!] = \mathsf{GC}(\mathcal{C}[\![\Gamma]\!])$. If $\Gamma = \Gamma', x : A$, then $\mathcal{C}[\![\Gamma', x : A]\!] = \mathcal{C}[\![\Gamma']\!] \times [\![A]\!]$. By induction $\mathcal{C}[\![\Gamma']\!] \cong \mathsf{GC}(\mathcal{C}[\![\Gamma']\!])$. By definition, Lemma 5.8 $[\![A]\!] \cong \mathsf{GC}([\![A]\!])$. Hence, $\mathcal{C}[\![\Gamma', x : A]\!] \cong \mathsf{GC}(\mathcal{C}[\![\Gamma', x : A]\!])$. □

Lemma 5.12. *Given a syntactic context Γ, Γ' such that Γ' is tick-free, there is a restriction morphism*

$$(-)|_{\Gamma} : \mathcal{C}[\![\Gamma, \Gamma']\!] \to \mathcal{C}[\![\Gamma]\!]$$

Proof. We proceed by induction on the length of Γ'. If $\Gamma' = \cdot$ the conclusion follows trivially. If $\Gamma' = \Gamma'', x : A$, the conclusion follows by application of the induction hypothesis and by projecting away $[\![A]\!](w)$. □

Definition 5.13. We define the morphism $\mathsf{untick} : \Downarrow \sqcup A \to \sqcup A$

$$\mathsf{untick} = \lambda w. \begin{cases} (-)|_w & \downarrow w \text{ defined} \\ ! & \downarrow w \text{ not defined} \end{cases}$$

To see why this is well-defined, note first that if $\downarrow w$ is not defined, then $(\Downarrow \sqcup A)(w) = \emptyset$. In the case that $\downarrow w$ is defined, we know that $w = ((\eta \checkmark \eta'), \overline{\eta}, \alpha)$. By definition, we thus have

$$(\Downarrow(\sqcup A))((\eta \checkmark \eta'), \overline{\eta}, \alpha) = \sqcup(A)(\eta, (\eta'; \overline{\eta}), \alpha + 1) = \operatorname*{colim}_{\overline{\eta}'} A(\bullet, \overline{\eta}', \alpha + 1)$$

On the other hand we have

$$(\sqcup A)((\eta \checkmark \eta'), \overline{\eta}, \alpha) = \operatorname*{colim}_{\overline{\eta}'} A(\bullet, \overline{\eta}', \alpha)$$

and hence

$$(-)|_w : \operatorname*{colim}_{\overline{\eta}'} A(\bullet, \overline{\eta}', \alpha + 1) \to \operatorname*{colim}_{\overline{\eta}'} A(\bullet, \overline{\eta}', \alpha)$$

as wanted.

Lemma 5.14. *Given a syntactic context $\Gamma, \sharp, \Gamma'$, there is a restriction morphism*

$$(-)|_{\Gamma,\sharp} : \mathcal{C}[\![\Gamma, \sharp, \Gamma']\!] \to \mathcal{C}[\![\Gamma, \sharp]\!]$$

Proof. In the case that the interpretation is empty, the conclusion follow trivially. Assume now that the interpretation is non-empty. We proceed by induction over Γ'. If $\Gamma' = \cdot$, then the map is just the identity. If $\Gamma' = \Gamma'', x : A$, then $\mathcal{C}[\![\Gamma, \sharp, \Gamma', x : A]\!] = \mathcal{C}[\![\Gamma, \sharp, \Gamma']\!] \times [\![A]\!]$. By application of the projection we get $\mathcal{C}[\![\Gamma, \sharp, \Gamma']\!]$ then by induction $\mathcal{C}[\![\Gamma, \sharp]\!]$. If $\Gamma' = \Gamma'', \checkmark$, then $\mathcal{C}[\![\Gamma, \sharp, \Gamma', \checkmark]\!] = \Downarrow \mathcal{C}[\![\Gamma, \sharp, \Gamma']\!]$. By application of the induction hypothesis under $\Downarrow$ we get $\Downarrow \mathcal{C}[\![\Gamma, \sharp]\!]$. By definition this is $\Downarrow \sqcup \mathcal{C}[\![\Gamma]\!]$ and then by untick we get $\sqcup \mathcal{C}[\![\Gamma]\!] = \mathcal{C}[\![\Gamma, \sharp]\!]$ as wanted. □

Definition 5.15. We define the map $\mathsf{run} : \sqcup \mathcal{T}(A) \to \sqcup A$

$$\mathsf{run} = \lambda w. \begin{cases} [\lambda t. t(\bullet, \overline{\eta}', w.\alpha)(*)]_{\overline{\eta}'} & w.\sigma \neq \bullet \\ ! & w.\sigma = \bullet \end{cases}$$

To see why this is well-defined, note first that in the case of $w.\sigma = \bullet$, then $(\sqcup \mathcal{T}(A))(w) = \emptyset$. If $w.\sigma \neq \bullet$, then

$$(\sqcup \mathcal{T}(A))(w) = \operatorname*{colim}_{\overline{\eta}'} \mathcal{T}(A)(\bullet, \overline{\eta}', \alpha)$$

and hence, to give a map out of the colimit, it is sufficient to give a map for each $\overline{\eta}'$ out of $\mathcal{T}(A)(\bullet, \overline{\eta}', \alpha)$ since we can then apply a coparing. Assume some $\overline{\eta}'$ and $t \in \mathcal{T}(A)(\bullet, \overline{\eta}', \alpha)$, by definition of the term interpretation, we then have

$$t(\bullet, \overline{\eta}', \alpha) : \mathcal{S}((\bullet, \overline{\eta}', \alpha), (\bullet, \overline{\eta}', \alpha)) \to (\sqcup A)(\bullet, \overline{\eta}', \alpha)$$

but by definition of the store interpretation this is just

$$1 \to (\sqcup A)(\bullet, \overline{\eta}', \alpha)$$

and hence, we see

$$t(\bullet, \overline{\eta}', \alpha)(*) : (\sqcup A)(\bullet, \overline{\eta}', \alpha)$$

as wanted.

Definition 5.16. We define the morphism $\mathsf{Fix} : \sqcap(A^{\Uparrow A}) \to \sqcap A$. Given a world w we first split into cases based on whether $w.\sigma = \bullet$ and in that case, we proceed by recursion on $w.\alpha$:

$$\mathsf{Fix} = \lambda w.\lambda f. \begin{cases} ! & w.\sigma \neq \bullet \\ \langle (\pi_{\overline{\eta}}(f))_{(\emptyset,\overline{\eta},0)}(*)\rangle_{\overline{\eta}} & w.\alpha = 0 \\ \langle (\pi_{(\eta;\overline{\eta})}(f))_{(\emptyset,(\eta;\overline{\eta}),\alpha')}(\pi_{\overline{\eta}}(\mathsf{Fix}(\bullet,(\eta;\overline{\eta}),\alpha')(f_{(\bullet,(\eta;\overline{\eta}),\alpha')})))_{(\emptyset\checkmark\eta,\overline{\eta},\alpha')}\rangle_{(\eta;\overline{\eta})} & w.\alpha = \alpha' + 1 \end{cases}$$

To see why this is well-defined, note first that if $w.\sigma \neq \bullet$ then by definition we have $(\sqcap A)(w) = \mathsf{1}$ and hence $! : (\sqcap(A^{\Uparrow A}))(w) \to (\sqcap A)(w)$ as wanted. If $w.\sigma = \bullet$ then

$$(\sqcap A)(w) = \varprojlim_{\overline{\eta}} A(\emptyset, \overline{\eta}, w.\alpha)$$

To give a map into $(\sqcap A)(w)$ it is sufficient to give a map into $A(\emptyset, \overline{\eta}, w.\alpha)$ for all $\overline{\eta}$, since we can then pair up all of these. To give this map, we proceed by recursion over $w.\alpha$:

- $\alpha = 0$:
  Assume $f : (\sqcap(A^{\Uparrow A}))(\bullet, w.\overline{\eta}, 0) = \varprojlim_{\overline{\eta}}(A^{\Uparrow A})(\emptyset, \overline{\eta}, 0)$. Assume now some $\overline{\eta}$. We see that $\pi_{\overline{\eta}}(f) : (A^{\Uparrow A})(\emptyset, \overline{\eta}, 0)$. By definition $(\Uparrow A)(\emptyset, \overline{\eta}', 0) = \mathsf{1}$, and we thus see that

  $$(\pi_{\overline{\eta}}(f_{(\emptyset,\overline{\eta},0)}))(*) : A(\emptyset, \overline{\eta}', 0)$$

  as wanted.

- $\alpha = \alpha' + 1$:
  Assume $f : (\sqcap(A^{\Uparrow A}))(\bullet, w.\overline{\eta}, \alpha' + 1)$. Assume some $(\eta; \overline{\eta})$ and we see that

  $$(\pi_{(\eta;\overline{\eta})}(f))_{(\emptyset,(\eta;\overline{\eta}),\alpha'+1)} : (\Uparrow A)(\emptyset, (\eta;\overline{\eta}), \alpha' + 1) \to A(\emptyset, (\eta;\overline{\eta}), \alpha' + 1)$$

  By definition we see

  $$(\Uparrow A)(\emptyset, (\eta;\overline{\eta}), \alpha' + 1) = A(\emptyset\checkmark\eta, \overline{\eta}'', \alpha')$$

  By recursion on α', we now see that

  $$\mathsf{Fix}(\bullet, (\eta;\overline{\eta}), \alpha') : (\sqcap(A^{\Uparrow A}))(\bullet, \overline{\eta}, \alpha') \to (\sqcap A)(\bullet, (\eta;\overline{\eta}), \alpha')$$

  By weakening we have $f_{(\bullet,(\eta;\overline{\eta}),\alpha')} : (\sqcap(A^{\Uparrow A}))(\bullet, (\eta;\overline{\eta}), \alpha')$ and hence

  $$(\mathsf{Fix}(\bullet, (\eta;\overline{\eta}), \alpha'))(f_{(\bullet,(\eta;\overline{\eta}),\alpha')}) : (\sqcap A)(\bullet, (\eta;\overline{\eta}), \alpha')$$

  We can now do a projection out of the limit and a weakening to get

  $$\pi_{\overline{\eta}}(\mathsf{Fix}(\bullet, (\eta;\overline{\eta}), \alpha'))(f_{(\bullet,(\eta;\overline{\eta}),\alpha')}) : A(\emptyset\checkmark\eta, \overline{\eta}, \alpha')$$

  Combining this with the above we thus get

  $$(\pi_{(\eta;\overline{\eta})}(f))_{(\emptyset,(\eta;\overline{\eta}),\alpha'+1)}(\pi_{\overline{\eta}}(\mathsf{Fix}(\bullet, (\eta;\overline{\eta}), \alpha'))(f_{(\bullet,(\eta;\overline{\eta}),\alpha')})) : A(\emptyset, (\eta;\overline{\eta}), \alpha' + 1)$$

  as wanted.

Lemma 5.17. *If A stable then for any store σ, any pair of heap sequences $\overline{\eta}, \overline{\eta}'$ and any ordinal α there is a morphism*

$$[\![A]\!](\bullet, \overline{\eta}, \alpha) \to [\![A]\!](\sigma, \overline{\eta}', \alpha)$$

Proof. We proceed by induction over A. For $\mathsf{1}$ and Nat the conclusion follows since they are both constant. For $A \times B$ and $A + B$ it follows by application of the induction hypothesis. For $\Box A$ it follows by definition note that the right hand side is either equal to the left hand side or the terminal object. Hence, the map is either the identity or the unique map into the terminal object. □

Definition 5.18. Given a syntactic context $\Gamma, x : A, \Gamma'$ s.t. either Γ' is tokenfree or $A : \mathsf{stable}$, then we define the projection map $\pi_A : \mathcal{C}[\![\Gamma, x : A, \Gamma']\!] \to [\![A]\!]$ by cases.

- Γ' tokenfree:
  In this case, we see that by application of the restriction map of Lemma 5.12 we get to $\mathcal{C}[\![\Gamma, x : A]\!] = \mathcal{C}[\![\Gamma]\!] \times [\![A]\!]$ and the by a projection we get to $[\![A]\!]$.
- $\Gamma' = \Gamma_1, \sharp, \Gamma_2$ and A stable :
  In this case, we see that by application of the restriction map from Lemma 5.14 we get to $\mathcal{C}[\![\Gamma, x : A, \Gamma_1, \sharp]\!] = \sqcup\, \mathcal{C}[\![\Gamma, x : A, \Gamma_1]\!]$. We can then apply the restriction from Lemma 5.12 under $\sqcup$ to get to $\sqcup\, \mathcal{C}[\![\Gamma, x : A]\!]$. We now assume some world w. If $w.\sigma = \bullet$ then $(\sqcup\, \mathcal{C}[\![\Gamma, x : A]\!])(w) = \emptyset$ and hence we define the map as the unique map out of the empty set. If $w.\sigma \neq \bullet$ then

  $$(\sqcup\, \mathcal{C}[\![\Gamma, x : A]\!])(w) = \operatorname*{colim}_{\overline{\eta}} \mathcal{C}[\![\Gamma, x : A]\!](\bullet, \overline{\eta}, w.\alpha)$$

  To give a map out of colimit, it is sufficient to give it for all $\overline{\eta}$, since we can then do the copairing of these. Assume $\overline{\eta}$ we can now apply a projection to get to $[\![A]\!](\bullet, \overline{\eta}, w.\alpha)$ and then by Lemma 5.17 we get to $[\![A]\!](w)$ as wanted.

5.4 Term Interpretations

Definition 5.19. We define the interpretation of well-typed terms by induction over the typing tree. Given $\Gamma \vdash t : A$, the interpretation is a morphism from $\mathcal{C}[\![\Gamma]\!]$ to $\mathcal{T}([\![A]\!])$. We will write $[\![t]\!]$ for

$[\![\Gamma \vdash t : A]\!]$ when the context is clear.

$$
\begin{aligned}
[\![\Gamma, x : A, \Gamma' \vdash x : A]\!](\gamma) &= (\eta_{\mathcal{T}} \circ \pi_A)(\gamma) \\
[\![\Gamma \vdash \langle\rangle : 1]\!](\gamma) &= (\eta_{\mathcal{T}} \circ !_{1_{\mathcal{R}}})(\gamma) \\
[\![\Gamma \vdash \overline{n} : \mathsf{Nat}]\!](\gamma) &= (\eta_{\mathcal{T}} \circ \Delta)(n) \\
[\![\Gamma \vdash s + t : \mathsf{Nat}]\!](\gamma) &= (\mathcal{T}(\mathsf{plus}) \circ \mathsf{comm} \circ \langle [\![s]\!], [\![t]\!] \rangle)(\gamma) \\
[\![\Gamma \vdash \lambda x.t : A \to B]\!](\gamma) &= (\eta_{\mathcal{T}} \circ \mathsf{GC}(\mathsf{curry}([\![t]\!])))(\gamma) \\
[\![\Gamma \vdash t\, t' : B]\!](\gamma) &= (\mu_{\mathcal{T}} \circ \mathcal{T}(\mathsf{ev}) \circ \mathsf{comm} \circ \langle \mathcal{T}(\varepsilon_{\mathsf{GC}}) \circ [\![t]\!], [\![t']\!] \rangle)(\gamma) \\
[\![\Gamma \vdash \langle s, t\rangle : A \times B]\!](\gamma) &= (\mathsf{comm} \circ \langle [\![s]\!], [\![t]\!] \rangle)(\gamma) \\
[\![\Gamma \vdash \pi_i\, t : A_i]\!](\gamma) &= (\mathcal{T}(\pi_i) \circ [\![t]\!])(\gamma) \\
[\![\Gamma \vdash \mathsf{in}_i\, t : A_1 + A_2]\!](\gamma) &= (\mathcal{T}(\mathsf{in}_i\, t) \circ [\![t]\!])(\gamma) \\
[\![\Gamma \vdash \mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2 B :]\!](\gamma) &= (\mu_{\mathcal{T}} \circ \mathcal{T}(\mathsf{ev}) \circ \mathsf{comm} \circ \langle \eta_{\mathcal{T}} \circ \langle \mathsf{curry}([\![t_1]\!]), \mathsf{curry}([\![t_2]\!]) \rangle, [\![t]\!] \rangle)(\gamma) \\
[\![\Gamma \vdash \mathsf{delay}\, t : \bigcirc t]\!](\gamma) &= (\eta_{\mathcal{T}} \circ \tau_{\Downarrow\Uparrow} [\![t]\!])(\gamma) \\
[\![\Gamma, \checkmark, \Gamma' \vdash \mathsf{adv}\, t : A]\!](\gamma) &= (\varepsilon_{\Downarrow\Uparrow} \circ \mathsf{step} \circ \Downarrow [\![t]\!])(\gamma|_{\Gamma,\checkmark}) \\
[\![\Gamma, \sharp, \Gamma' \vdash \mathsf{unbox}\, t : A]\!](\gamma) &= (\varepsilon_{\sqcup\sqcap} \circ \mathsf{run} \circ \sqcup [\![t]\!])(\gamma|_{\Gamma,\sharp}) \\
[\![\Gamma \vdash \mathsf{box}\, t : \Box A]\!](\gamma) &= (\eta_{\mathcal{T}} \circ \tau_{\sqcup\sqcap} [\![t]\!])(\gamma) \\
[\![\Gamma \vdash \mathsf{fix}\, \alpha.t : \Box A]\!](\gamma) &= (\eta_{\mathcal{T}} \circ \mathsf{Fix} \circ \tau_{\sqcup\sqcap}(\mathsf{curry}([\![t]\!])))(\gamma) \\
[\![\Gamma \vdash \mathsf{into}\, t : \mu\alpha.A]\!](\gamma) &= [\![t]\!](\gamma) \\
[\![\Gamma \vdash \mathsf{out}\, t : A[\bigcirc \mu\alpha.A/x]]\!](\gamma) &= [\![t]\!](\gamma)
\end{aligned}
$$

To see that the above is well-defined, we go through them one by one.

- $[\![\Gamma, x : A, \Gamma \vdash x : A]\!](\gamma)$:
  To see why this is well-defined, consider the following diagram

$$\mathcal{C}[\![\Gamma, x : A, \Gamma']\!] \overset{\pi_A}{\to} [\![A]\!] \overset{\eta_{\mathcal{T}}}{\to} \mathcal{T}([\![A]\!])$$

- $[\![\Gamma \vdash \langle\rangle : 1]\!](\gamma)$:
  To see why this is well-defined, consider the following diagram

$$\mathcal{C}[\![\Gamma]\!] \overset{!_{1_{\mathcal{R}}}}{\to} [\![1]\!] \overset{\eta_{\mathcal{T}}}{\to} \mathcal{T}([\![1]\!])$$

- $[\![\Gamma \vdash \overline{n} : \mathsf{Nat}]\!](\gamma)$:
  To see why this is well-defined, note first that by assumption $n \in \mathbb{N}$ and now consider the following diagram

$$\mathbb{N} \overset{\Delta}{\to} \mathbb{N}_{\mathcal{R}} = [\![\mathsf{Nat}]\!] \overset{\eta_{\mathcal{T}}}{\to} \mathcal{T}([\![\mathsf{Nat}]\!])$$

- $[\![\Gamma \vdash n + m : \mathsf{Nat}]\!](\gamma)$:
  To see why this is well-defined, consider the following digram

$$\mathcal{C}[\![\Gamma]\!] \overset{\langle [\![n]\!], [\![t]\!] \rangle}{\to} \mathcal{T}([\![\mathsf{Nat}]\!]) \times \mathcal{T}([\![\mathsf{Nat}]\!]) \overset{\mathsf{comm}}{\to} \mathcal{T}([\![\mathsf{Nat}]\!] \times [\![\mathsf{Nat}]\!]) \overset{\mathcal{T}(\mathsf{plus})}{\to} \mathcal{T}([\![\mathsf{Nat}]\!])$$

- $[\![\Gamma \vdash \lambda x.t : A \to B]\!](\gamma)$:
  By definition of the interpretation, we know

  $$[\![t]\!] : \mathcal{C}[\![\Gamma, x : A]\!] \to \mathcal{T}([\![B]\!])$$

  By definition $\mathcal{C}[\![\Gamma, x : A]\!] = \mathcal{C}[\![\Gamma]\!] \times [\![A]\!]$. We thus have

  $$\mathsf{curry}([\![t]\!]) : \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}([\![B]\!])^{[\![A]\!]}$$

  and then

  $$\mathsf{GC}(\mathsf{curry}([\![t]\!])) : \mathsf{GC}(\mathcal{C}[\![\Gamma]\!]) \to \mathsf{GC}(\mathcal{T}([\![B]\!])^{[\![A]\!]})$$

  Since Γ is assumed to be tick-free, we have by Lemma 5.11 $\mathcal{C}[\![\Gamma]\!] \cong \mathsf{GC}(\mathcal{C}[\![\Gamma]\!])$. Finally, by definition of the value interpretation $\mathsf{GC}(\mathcal{T}([\![B]\!])^{[\![A]\!]}) = [\![A \to B]\!]$ and thus by composing with $\eta_{\mathcal{T}}$ we get to $\mathcal{T}([\![A \to B]\!])$ as wanted.

- $[\![\Gamma \vdash t\, t' : B]\!](\gamma)$:
  By definition of the interpretation, we know

  $$[\![t]\!] : \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}(\mathsf{GC}(\mathcal{T}([\![B]\!])^{[\![A]\!]}))$$

  and

  $$[\![t']\!] : \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}([\![A]\!])$$

  We thus have

  $$\mathcal{T}(\varepsilon_{\mathsf{GC}})([\![t]\!](\gamma)) : \mathcal{T}(\mathcal{T}([\![B]\!])^{[\![A]\!]})$$

  and further

  $$\langle \mathcal{T}(\varepsilon_{\mathsf{GC}})([\![t]\!], [\![t']\!] \rangle : \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}(\mathcal{T}([\![B]\!])^{[\![A]\!]}) \times \mathcal{T}([\![A]\!])$$

  By composing with comm we get to $\mathcal{T}(\mathcal{T}([\![B]\!])^{[\![A]\!]} \times [\![A]\!])$ and then by evaluation under the term interpretation, we get to $\mathcal{T}(\mathcal{T}([\![B]\!]))$. Finally, by term multiplication we get to $\mathcal{T}([\![B]\!])$ as wanted.

- $[\![\Gamma \vdash \langle s, t \rangle : A \times B]\!](\gamma)$:
  To see why this is well-defined, consider the following diagram

  $$\mathcal{C}[\![\Gamma]\!] \overset{\langle [\![s]\!], [\![t]\!] \rangle}{\to} \mathcal{T}([\![A]\!]) \times \mathcal{T}([\![B]\!]) \overset{\mathsf{comm}}{\to} \mathcal{T}([\![A]\!] \times [\![B]\!])$$

- $[\![\Gamma \vdash \pi_i\, t : A_i]\!](\gamma)$:
  To see why this is well-defined, consider the following diagram

  $$\mathcal{C}[\![\Gamma]\!] \overset{[\![t]\!]}{\to} \mathcal{T}([\![A_1 \times A_2]\!]) = \mathcal{T}([\![A_1]\!] \times [\![A_2]\!]) \overset{\mathcal{T}(\pi_i)}{\to} \mathcal{T}([\![A_i]\!])$$

- $[\![\Gamma \vdash \mathsf{in}_i\, t : A_1 + A_2]\!](\gamma)$:
  To see why this is well-defined, consider the following diagram

  $$\mathcal{C}[\![\Gamma]\!] \overset{[\![t]\!]}{\to} \mathcal{T}([\![A_i]\!]) \overset{\mathcal{T}(\mathsf{in}_i)}{\to} \mathcal{T}([\![A_1]\!] + [\![A_2]\!]) = \mathcal{T}([\![A_1 + A_2]\!])$$

- $[\![\Gamma \vdash \mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2 B :]\!](\gamma)$:
  We note first that $[\![t]\!] : \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}([\![A_1 + A_2]\!])$ and $[\![t_i]\!] : \mathcal{C}[\![\Gamma]\!] \times [\![A_i]\!] \to \mathcal{T}([\![B]\!])$. Hence, we have $\mathsf{curry}([\![t_i]\!]) : \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}([\![B]\!])^{[\![A_i]\!]}$. By pairing up these we get

  $$\langle \mathsf{curry}([\![t_1]\!]), \mathsf{curry}([\![t_2]\!]) \rangle : \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}([\![B]\!])^{[\![A_1]\!]} \times \mathcal{T}([\![B]\!])^{[\![A_2]\!]}$$

  and since $\mathcal{R}$ is distributive, we have $\mathcal{T}([\![B]\!])^{[\![A_1]\!]} \times \mathcal{T}([\![B]\!])^{[\![A_2]\!]} \cong \mathcal{T}([\![B]\!])^{[\![A_1]\!]+[\![A_2]\!]}$ and by definition $[\![A_1]\!] + [\![A_2]\!] = [\![A_1 + A_2]\!]$. Further, we now see that

  $$\eta_{\mathcal{T}} \circ \langle \mathsf{curry}([\![t_1]\!]), \mathsf{curry}([\![t_2]\!]) \rangle : \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}(\mathcal{T}([\![B]\!])^{[\![A_1+A_2]\!]})$$

  Pairing up this map with $[\![t]\!]$ and commuting we thus get a map

  $$\mathcal{C}[\![\Gamma]\!] \to \mathcal{T}(\mathcal{T}([\![B]\!])^{[\![A_1+A_2]\!]} \times [\![A_1 + A_2]\!])$$

  and thus by evaluating under the term interpretation, we get to $\mathcal{T}(\mathcal{T}([\![B]\!]))$ and by the multiplication of the term interpretation we get to $\mathcal{T}([\![B]\!])$ as wanted.

- $[\![\Gamma \vdash \mathsf{delay}\, t : \bigcirc A]\!](\gamma)$:
  We see that $[\![t]\!] : \mathcal{C}[\![\Gamma, \checkmark]\!] \to \mathcal{T}([\![A]\!])$. By definition of the context interpretation, $\mathcal{C}[\![\Gamma, \checkmark]\!] = \Downarrow \mathcal{C}[\![\Gamma]\!]$, and hence

  $$[\![t]\!] : \Downarrow \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}([\![A]\!])$$

  By definition of the $\Downarrow \dashv \Uparrow$ adjunction, we thus have

  $$\tau_{\Downarrow\Uparrow}([\![t]\!]) : \mathcal{C}[\![\Gamma]\!] \to \Uparrow \mathcal{T}([\![A]\!])$$

  By definition of the value interpretation, $\Uparrow \mathcal{T}([\![A]\!]) = [\![\bigcirc A]\!]$ and hence by composing with $\eta_{\mathcal{T}}$ we get $\mathcal{T}([\![\bigcirc A]\!])$ as wanted.

- $[\![\Gamma, \checkmark, \Gamma' \vdash \mathsf{adv}\, t : A]\!](\gamma)$:

  To see why this is well-defined, note first that $\mathcal{C}[\![\Gamma, \checkmark, \Gamma']\!] \overset{|_{\Gamma,\checkmark}}{\to} \mathcal{C}[\![\Gamma, \checkmark]\!]$ and by definition, $\mathcal{C}[\![\Gamma, \checkmark]\!] = \Downarrow \mathcal{C}[\![\Gamma]\!]$. By assumption, we have

  $$[\![t]\!] : \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}(\bigcirc A)$$

  and thus

  $$\Downarrow([\![t]\!]) : \Downarrow \mathcal{C}[\![\Gamma]\!] \to \Downarrow \mathcal{T}(\bigcirc A)$$

  By definition we have $\Downarrow \mathcal{T}(\bigcirc A) = \Downarrow \mathcal{T}(\Uparrow \mathcal{T}([\![A]\!]))$. Further we see

  $$\Downarrow \mathcal{T}(\Uparrow \mathcal{T}([\![A]\!])) \overset{\mathsf{step}}{\to} \Downarrow \Uparrow \mathcal{T}([\![A]\!])$$

  and finally $\Downarrow \Uparrow \mathcal{T}([\![A]\!]) \overset{\varepsilon_{\Downarrow\Uparrow}}{\to} \mathcal{T}([\![A]\!])$ as wanted.

- $[\![\Gamma, \sharp, \Gamma' \vdash \mathsf{unbox}\, t : A]\!](\gamma)$:

  To see why this is well-defined, note first that we have the restriction map $\mathcal{C}[\![\Gamma, \sharp, \Gamma']\!] \overset{|_{\Gamma,\sharp}}{\to} \mathcal{C}[\![\Gamma, \sharp]\!]$. By definition of the context interpretation we know $\mathcal{C}[\![\Gamma, \sharp]\!] = \sqcup \mathcal{C}[\![\Gamma]\!]$. By assumption we have $[\![t]\!] : \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}([\![\Box A]\!])$, and hence $\sqcup [\![t]\!] : \sqcup \mathcal{C}[\![\Gamma]\!] \to \sqcup \mathcal{T}([\![\Box A]\!])$. By definition of the value interpretation, $\sqcup \mathcal{T}([\![\Box A]\!]) = \sqcup \mathcal{T}(\sqcap \mathcal{T}([\![A]\!]))$. By application of the run morphism we then get to $\sqcup \sqcap \mathcal{T}(A)$ and then by the counit of the $\sqcup \dashv \sqcap$ adjunction, we get to $\mathcal{T}(A)$ as wanted.

- $[\![\Gamma \vdash \mathsf{box}\, t : \Box A]\!](\gamma)$:
  We see that $[\![t]\!] : \mathcal{C}[\![\Gamma, \sharp]\!] \to \mathcal{T}(A)$. By definition of the context interpretation, this is equally $\sqcup \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}([\![A]\!])$. By definition of the $\sqcup \dashv \sqcap$ adjunction, we thus have

  $$\tau_{\sqcup \sqcap}([\![t]\!]) : \mathcal{C}[\![\Gamma]\!] \to \sqcap \mathcal{T}([\![A]\!])$$

  By definition of the value interpretation, $\sqcap \mathcal{T}([\![A]\!]) = [\![\Box A]\!]$ and we thus see that by composing with $\eta_{\mathcal{T}}$ we get to $\mathcal{T}([\![\Box A]\!])$ as wanted.

- $[\![\Gamma \vdash \mathsf{fix}\, \alpha.t : \Box A]\!](\gamma)$:
  By assumption we have

  $$[\![t]\!] : \mathcal{C}[\![\Gamma, \sharp, \alpha : \bigcirc A]\!] \to \mathcal{T}([\![A]\!])$$

  By definition of the context interpretation

  $$\mathcal{C}[\![\Gamma, \sharp, \alpha : \bigcirc A]\!] = \sqcup \mathcal{C}[\![\Gamma]\!] \times \Uparrow \mathcal{T}([\![A]\!])$$

  Hence we have

  $$[\![t]\!] : \sqcup \mathcal{C}[\![\Gamma]\!] \times \Uparrow \mathcal{T}([\![A]\!]) \to \mathcal{T}([\![A]\!])$$

  We then see that

  $$\mathsf{curry}([\![t]\!]) : \sqcup \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}([\![A]\!])^{\Uparrow \mathcal{T}(A)}$$

  By definition of the $\sqcup \dashv \sqcap$ adjunction, we thus get

  $$\tau_{\sqcup \sqcap}(\mathsf{curry}([\![t]\!])) : \mathcal{C}[\![\Gamma]\!] \to \sqcap(\mathcal{T}([\![A]\!])^{\Uparrow \mathcal{T}([\![A]\!])})$$

  We now see that

  $$\mathsf{Fix} \circ \tau_{\sqcup \sqcap}(\mathsf{curry}([\![t]\!])) : \mathcal{C}[\![\Gamma]\!] \to \sqcap \mathcal{T}([\![A]\!])$$

  By definition $\sqcap \mathcal{T}([\![A]\!]) = [\![\Box A]\!]$ and hence by composing with $\eta_{\mathcal{T}}$ we get to $\mathcal{T}([\![\Box A]\!])$ as wanted.

- $[\![\Gamma \vdash \mathsf{into}\, t : \mu\alpha.A]\!](\gamma)$:
  By assumption we have $[\![t]\!] : \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}([\![A[\bigcirc \mu\alpha.A/\alpha]]\!])$ but by definition $[\![A[\bigcirc \mu\alpha./\alpha]]\!] = [\![\mu\alpha.A]\!]$ and hence $[\![t]\!] : \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}([\![\mu\alpha.A]\!])$ as wanted.

- $[\![\Gamma \vdash \mathsf{out}\, t : A[\bigcirc \mu\alpha.A/\alpha]]\!]$:
  By assumption we have $[\![t]\!] : \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}([\![\mu\alpha.A]\!])$ but by definition $[\![\mu\alpha.A]\!] = [\![A[\bigcirc \mu\alpha./\alpha]]\!]$ and hence $[\![t]\!] : \mathcal{C}[\![\Gamma]\!] \to \mathcal{T}([\![A[\bigcirc \mu\alpha.A/\alpha]]\!])$ as wanted.

6 Conclusion And Future Work

In this manuscript, we have presented categorical semantics for Simply RaTT. In particular, we have shown that both $\Box$ and $\bigcirc$ are interpreted using pairs of adjoint functors. Further, we show

how values are co-algebras over an idempotent comonad and how terms are given by a commutative reader like monad.

In future work, we would like to investigate the operational properties of Simply RaTT in the model. In particular, we want to encode streams and do a "step semantics" as in [BGM19]. It should be possible prove that these streams are also free of spaceleaks and in that work.

Finally, since the model is a presheaf, and hence a topos, we are interested in the looking at the internal logic with a view towards dependent types.

References

[BGM19] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. Simply ratt: a fitch-style modal calculus for reactive programming without space leaks. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–27, 2019.

[BGM20] Patrick Bahr, Christian Uldal Graulund, and Rasmus Møgelberg. Diamonds are not forever: Liveness in reactive programming with guarded recursion, 2020.

[Bor94] Francis Borceux. *Handbook of Categorical Algebra*, volume 2 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 1994.

[CFPP14] Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. Fair Reactive Programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 361–372, San Diego, California, USA, 2014. ACM.

[Clo18] Ranald Clouston. Fitch-style modal lambda calculi. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 258–275, Cham, 2018. Springer International Publishing.

[CMM+18] Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. Modal dependent type theory and dependent right adjoints. *CoRR*, abs/1804.05236:1–21, 2018.

[EH97] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM.

[Fit52] F. B Fitch. *Symbolic logic, an introduction*. Ronald Press Co., New York, NY, USA, 1952.

[Jef12] Alan Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In Koen Claessen and Nikhil Swamy, editors, *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*, pages 49–60, Philadelphia, PA, USA, 2012. ACM.

[Jef14] Alan Jeffrey. Functional reactive types. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 54:1–54:9, New York, NY, USA, 2014. ACM.

[Kri13] Neelakantan R. Krishnaswami. Higher-order Functional Reactive Programming Without Spacetime Leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 221–232, Boston, Massachusetts, USA, 2013. ACM.

[MM18] Bassel Mannaa and Rasmus Ejlers Møgelberg. The clocks they are adjunctions denotational semantics for clocked type theory. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPIcs*, pages 23:1–23:17, New York, NY, USA, 2018. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[Nak00] Hiroshi Nakano. A modality for recursion. *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pages 255–266, 2000.

Chapter 4

Paper 2: Diamonds are Not Forever: Liveness in Reactive Programming with Guarded Recursion

Diamonds Are Not Forever

Liveness in Reactive Programming with Guarded Recursion

PATRICK BAHR, IT University of Copenhagen, Denmark
CHRISTIAN ULDAL GRAULUND, IT University of Copenhagen, Denmark
RASMUS EJLERS MØGELBERG, IT University of Copenhagen, Denmark

When designing languages for functional reactive programming (FRP) the main challenge is to provide the user with a simple, flexible interface for writing programs on a high level of abstraction while ensuring that all programs can be implemented efficiently in a low-level language. To meet this challenge, a new family of modal FRP languages has been proposed, in which variants of Nakano's guarded fixed point operator are used for writing recursive programs guaranteeing properties such as causality and productivity. As an apparent extension to this it has also been suggested to use Linear Temporal Logic (LTL) as a language for reactive programming through the Curry-Howard isomorphism, allowing properties such as termination, liveness and fairness to be encoded in types. However, these two ideas are in conflict with each other, since the fixed point operator introduces non-termination into the inductive types that are supposed to provide termination guarantees.

In this paper we show that by regarding the modal time step operator of LTL a submodality of the one used for guarded recursion (rather than equating them), one can obtain a modal type system capable of expressing liveness properties while retaining the power of the guarded fixed point operator. We introduce the language Lively RaTT, a modal FRP language with a guarded fixed point operator and an 'until' type constructor as in LTL, and show how to program with events and fair streams. Using a step-indexed Kripke logical relation we prove operational properties of Lively RaTT including productivity and causality as well as the termination and liveness properties expected of types from LTL. Finally, we prove that the type system of Lively RaTT guarantees the absence of implicit space leaks.

CCS Concepts: • **Software and its engineering → Functional languages**; **Data flow languages**; *Recursion*; • **Theory of computation** → *Operational semantics*.

Additional Key Words and Phrases: Functional Reactive Programming, Modal Types, Linear Temporal Logic, Synchronous Data Flow Languages, Type Systems

ACM Reference Format:
Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2021. Diamonds Are Not Forever: Liveness in Reactive Programming with Guarded Recursion. *Proc. ACM Program. Lang.* 5, POPL, Article 2 (January 2021), 28 pages. https://doi.org/10.1145/3434283

1 INTRODUCTION

Reactive programs such as servers and control software in cars, aircrafts and robots are traditionally written in imperative languages using a wide range of complex features including call-backs and shared state. For this reason, they are notoriously error-prone and hard to reason about. This

Authors' addresses: Patrick Bahr, IT University of Copenhagen, Denmark, paba@itu.dk; Christian Uldal Graulund, IT University of Copenhagen, Denmark, cgra@itu.dk; Rasmus Ejlers Møgelberg, IT University of Copenhagen, Denmark, mogel@itu.dk.

2475-1421/2021/1-ART2
https://doi.org/10.1145/3434283

is unfortunate, since much of the most critical software currently in use is reactive. The goal of functional reactive programming (FRP) is to provide the programmer with tools for writing reactive programs on a high level of abstraction in the functional paradigm. In doing so, FRP extends the known benefits of functional programming also to reactive programming, in particular modularity and equational reasoning for programs. The challenge for achieving this goal is to ensure that all programs can be implemented efficiently in a low-level language.

From the outset, the central idea of FRP [Elliott and Hudak 1997] was that reactive programming simply is programming with signals and events. While elegant, this idea immediately leads to the question of what the interface for signals and events should be. A naive approach would be to model signals as streams in the sense of coinductive solutions to $\mathsf{Str}(A) \cong A \times \mathsf{Str}(A)$, but this allows the programmer to write *non-causal* programs, i.e., programs where the present output depends on future input. Arrowised FRP [Nilsson et al. 2002], as implemented in the Yampa library for Haskell, solves this problem by taking signal functions as primitive rather than signals themselves. However, this approach forfeits some of the simplicity of the original FRP model and reduces its expressivity as it rules out useful types such as signals of signals.

More recently, a number of authors [Bahr et al. 2019; Jeffrey 2014; Jeltsch 2013; Krishnaswami 2013; Krishnaswami and Benton 2011; Krishnaswami et al. 2012] have suggested a modal approach to FRP in which causality is ensured through the introduction of a notion of time in the form of a modal operator. In this approach, an element of the modal type $\triangleright A$ should be thought of as data of type A arriving in the next time step. Signals should be modelled as a type of streams satisfying the type isomorphism $\mathsf{Str}(A) \cong A \times \triangleright\mathsf{Str}(A)$ capturing the idea that each pair of elements of a stream is separated by a time step. Events carrying data of type A can be represented by a type satisfying $\mathsf{Ev}(A) \cong A + \triangleright\mathsf{Ev}(A)$, stating that an event can either occur now, or at some point in the future. Types such as $\mathsf{Str}(A)$ and $\mathsf{Ev}(A)$ satisfying type equations in which the recursion variable is guarded by a $\triangleright$ are referred to as *guarded recursive types*. Combining this with guarded recursion [Nakano 2000] in the form of a fixed point operator of type $(\triangleright A \to A) \to A$ gives a powerful type system for reactive programming guaranteeing not only causality, but also productivity, i.e. the property that for a closed stream, each of its elements can always be computed in finite time. In some systems [Bahr et al. 2019; Krishnaswami 2013; Krishnaswami et al. 2012] the modal types have also been used to guarantee the lack of implicit space leaks, i.e., the problem of programs holding on to memory while continually allocating until they run out of space. These leaks have previously been a major problem in FRP.

Jeffrey [2012] suggested taking this idea further using Linear Temporal Logic (LTL) [Pnueli 1977] as a type system for FRP through the Curry-Howard isomorphism, a connection discovered independently by Jeltsch [2012]. This idea is not only conceptually appealing, but could also extend the expressivity of the type system considerably and have practical consequences. Indeed, LTL has a step modality $\bigcirc$ similar to $\triangleright$ used to express that a formula should be true one time step from now. It also has an operation $\Box$ expressing global truth, i.e., formulas that hold now and at any time in the future. This operation has been used by Krishnaswami [2013] to express time-independent data that can be safely kept across time steps without causing space leaks. In this paper we are particularly interested in the *until* operator $\phi \,\mathcal{U}\, \psi$ of LTL, which expresses that ϕ holds now and for some more steps, after which ψ becomes true. Using this operator, we can encode the *finally* operator $\Diamond\phi$ as $\mathsf{tt} \,\mathcal{U}\, \phi$ stating that ϕ will eventually become true. In programming terms, the until operator is the inductive type given by constructors

$$\mathsf{now} : B \to A \,\mathcal{U}\, B \qquad\qquad \mathsf{wait} : A \to \bigcirc(A \,\mathcal{U}\, B) \to A \,\mathcal{U}\, B\,.$$

and the fact that it is inductive should imply a termination property similarly to that of LTL: Elements of type $A \,\mathcal{U}\, B$ will eventually produce a B after at most finitely many As, and similarly

for elements of type $\Diamond B$. In programming, this can be used to express the property that a program will eventually produce an output, e.g., by timeout, or one can give a type of fair schedulers [Cave et al. 2014], see section 3.2 for details.

The goal of this paper is to define a language combining the expressive type system of LTL with the power of the guarded recursive fixed point combinator. Unfortunately, equating $\bigcirc$ and $\triangleright$ in such a system breaks the termination guarantee of the $\mathcal{U}$ type. For example, if $a : A$, the fixed point of $\mathsf{wait}\, a : \bigcirc(A\ \mathcal{U}\ B) \to A\ \mathcal{U}\ B$ will never produce a B. This is an example of a well-known phenomenon: The guarded fixed point combinator implies uniqueness of solutions to guarded recursive type equations like $X \cong B + A \times \triangleright X$, and so inductive and coinductive solutions coincide. In fact, the solutions behave more like coinductive types than inductive types and can even be used to encode coinductive types [Atkey and McBride 2013] in some settings.

This observation led Cave et al. [2014] to suggest removing the guarded recursive fixed point operator from FRP in order to distinguish between inductive and coinductive guarded types. This has the unfortunate effect of losing the power and elegance of the guarded fixed point operator for programming with coinductive types, which ought to be safe. Indeed it is well known that programming directly with coiteration is cumbersome and so most programming languages allow the programmer to construct elements of coinductive types using recursion. To guarantee productivity, one must use either the (non-modular) syntactic checks used in most proof assistants today, or sized types [Abel and Pientka 2013; Abel et al. 2017; Hughes et al. 1996; Sacchini 2013]. Given that the modal operator is in the language, guarded recursion is the most obvious solution to guaranteeing productivity.

1.1 Overview of Results

In this paper we show that by considering $\bigcirc$ a submodality of $\triangleright$, rather than equating them, we can use the guarded fixed point operator while retaining the termination guarantees of $\mathcal{U}$. Using $\triangleright$, the type $\mathsf{Ev}(A)$ of possibly occurring events of type A can be encoded as the unique solution to $\mathsf{Ev}(A) \cong A + \triangleright\mathsf{Ev}(A)$. Using $\bigcirc$, the type $\Diamond A$ of events of type A that must occur can be encoded as above. We will often refer to these as the types of possibly non-terminating and terminating events, respectively. The inclusion from $\bigcirc$ into $\triangleright$ can be used to type an inclusion of $\Diamond A$ into $\mathsf{Ev}(A)$. The lack of an inclusion from $\triangleright$ to $\bigcirc$ means that there is no inclusion $\triangleright\Diamond A \to \Diamond A$ to take a fixed point of to construct a diverging element of $\Diamond A$.

To make these ideas concrete we define the language *Lively RaTT* (section 2) as an extension of the language Simply RaTT [Bahr et al. 2019]. Simply RaTT is an FRP language with modal operators $\triangleright$ and $\Box$ as described above, as well as guarded recursive types and guarded fixed points. It uses a Fitch-style approach [Clouston 2018; Clouston et al. 2018; Fitch 1952] to programming with modal types, which means that the typing rules for introduction and elimination for modal types add and remove tokens from a context. This gives a direct style for programming with modalities, avoiding let-expressions as traditionally used for elimination. Lively RaTT has tokens $\checkmark_{\triangleright}$ and $\checkmark_{\bigcirc}$ for $\triangleright$ and $\bigcirc$, respectively, and the inclusion of $\bigcirc$ into $\triangleright$ is defined by allowing $\checkmark_{\triangleright}$ to eliminate also $\bigcirc$. We think of $\checkmark_{\bigcirc}$ and $\checkmark_{\triangleright}$ as a separation in time in judgements: Variables to the left of $\checkmark_{\bigcirc}$ or $\checkmark_{\triangleright}$ are available one time step before those to the right. The token $\checkmark_{\triangleright}$ is a stronger time step, allowing also recursive definitions to be unfolded. We illustrate the expressivity of Lively RaTT by showing how to program with events and fair streams in section 3.

We define two kinds of operational semantics for Lively RaTT (section 4): An evaluation semantics reducing terms to values at each time instant, and a step semantics capturing the dynamic behaviour of reactive programs over time. The latter is defined for streams, $\mathcal{U}$-types, and fair streams only. We prove causality and productivity of streams, and we prove the termination property for $\mathcal{U}$-types, i.e., that any term of type $A\ \mathcal{U}\ B$ eventually produces a B, also in a context of a stream of external

$$\begin{aligned}
\textbf{Types} \quad & A, B ::= \alpha \mid 1 \mid \mathsf{Nat} \mid A \times B \mid A + B \mid A \to B \mid \Box A \mid \bigcirc A \mid \triangleright A \mid \mathsf{Fix}\ \alpha.A \mid A \mathbin{\mathcal{U}} B \\
\textbf{Stable types} \quad & S, S' ::= 1 \mid \mathsf{Nat} \mid S \times S' \mid S + S' \mid \Box A \\
\textbf{Limit types} \quad & L, L' ::= \alpha \mid 1 \mid \mathsf{Nat} \mid L \times L' \mid L + L' \mid A \to L \mid \Box L \mid \bigcirc L \mid \triangleright A \mid \mathsf{Fix}\ \alpha.L
\end{aligned}$$

Fig. 1. Grammars for types, stable types and limit types. In typing rules, only closed types are considered.

$$\frac{}{\cdot \vdash} \qquad \frac{\Gamma \vdash \quad x \notin \mathsf{dom}\,(\Gamma)}{\Gamma, x : A \vdash} \qquad \frac{\Gamma \vdash \quad \mathsf{lock\text{-}free}(\Gamma)}{\Gamma, \sharp \vdash} \qquad \frac{\Gamma \vdash \quad \sharp \in \Gamma \quad m \in \{\bigcirc, \triangleright\} \quad \mathsf{tick\text{-}free}(\Gamma)}{\Gamma, \checkmark_m \vdash}$$

Fig. 2. Well-formed contexts.

inputs. Using this, we prove that any term of the fair scheduler type can be unwound to a fair interleaving of streams, again also in a context of external input.

Finally, we show that the type system of Lively RaTT guarantees the lack of implicit space leaks. Our results on this extend those proved for Simply RaTT by Bahr et al. [2019] which in turn were based on a technique developed by [Krishnaswami 2013]. More precisely, our operational semantics stores input as well as delayed computations in a heap, and we show that it is safe to garbage collect the elements in the heap after two evaluation steps.

These results are proved (section 5) using an interpretation of types as sets of values indexed by four parameters, including an ordinal β. For finite β, this index should be thought of as a form of step-indexing: The interpretation of A at β in this case describes the behaviour of terms up to the first β evaluation steps. In our model, however, β runs all the way to $\omega \cdot 2$. The interpretation at higher β, in particular the limit ordinal ω describes global behaviour of programs.

The distinction between $\triangleright$ and $\bigcirc$ can be seen in the model. At successor ordinals $\beta + 1$, the interpretation of $\triangleright A$ and $\bigcirc A$ are both defined in terms of the interpretation of A at β in a step-indexed fashion [Birkedal et al. 2011], but at limit ordinals β, the interpretation of $\triangleright A$ is the intersection of the interpretations at $\beta' < \beta$, whereas $\bigcirc A$ is interpreted using the interpretation of A at β. This interpretation of $\triangleright A$ is needed to interpret fixed points, and the interpretation of $\bigcirc A$ ensures that the interpretation of $A \mathbin{\mathcal{U}} B$ behaves globally as an inductive type.

The paper ends with an overview of related work (section 6) and conclusions, perspectives and future work (section 7). Full proofs can be found in the accompanying technical report.

2 LIVELY RATT

Lively RaTT is an extension of Simply RaTT [Bahr et al. 2019], a Fitch-style modal language for reactive programming. This section gives an overview of the language, referring to Figure 3 for an overview of the typing rules.

In the Fitch-style approach to modal types the introduction and elimination rules for these add and remove tokens from a context. For example, the modality $\bigcirc$ expresses delay of data by one time step and has introduction and elimination rules as follows (ignoring $\triangleright$ for the moment).

$$\frac{\Gamma, \checkmark_\bigcirc \vdash t : A}{\Gamma \vdash \mathsf{delay}\, t : \bigcirc A} \qquad \frac{\Gamma \vdash t : \bigcirc A}{\Gamma, \checkmark_\bigcirc, \Gamma' \vdash \mathsf{adv}\, t : A}$$

The token $\checkmark_\bigcirc$ should be thought of as a separation by a single time step between the variables to the left of it and the rest of the judgement to the right. Thus the premise of the introduction rule

Simply typed λ-calculus:

$$\frac{\mathsf{token\text{-}free}(\Gamma') \vee A\ \mathsf{stable}}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{}{\Gamma \vdash \langle\rangle : 1} \qquad \frac{\Gamma, x : A \vdash t : B \qquad \mathsf{tick\text{-}free}(\Gamma)}{\Gamma \vdash \lambda x.t : A \to B}$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash t' : A}{\Gamma \vdash t\, t' : B} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash t' : B}{\Gamma \vdash \langle t, t'\rangle : A \times B} \qquad \frac{\Gamma \vdash t : A_1 \times A_2 \qquad i \in \{1,2\}}{\Gamma \vdash \pi_i\, t : A_i}$$

$$\frac{\Gamma \vdash t : A_i \qquad i \in \{1,2\}}{\Gamma \vdash \mathsf{in}_i\, t : A_1 + A_2} \qquad \frac{\Gamma, x : A_i \vdash t_i : B \qquad \Gamma \vdash t : A_1 + A_2 \qquad i \in \{1,2\}}{\Gamma \vdash \mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2 : B} \qquad \frac{}{\Gamma \vdash 0 : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash \mathsf{suc}\, t : \mathsf{Nat}} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma, x : \mathsf{Nat}, y : A \vdash t : A \qquad \Gamma \vdash n : \mathsf{Nat}}{\Gamma \vdash \mathsf{rec}_{\mathsf{Nat}}(s, x\, y.t, n) : A}$$

Modalities, $\mathcal{U}$-types, guarded recursion:

$$\frac{\Gamma, \checkmark_m \vdash t : A}{\Gamma \vdash \mathsf{delay}\, t : m\, A} \qquad \frac{\Gamma \vdash t : m\, A \qquad m \leqslant m' \vee A\ \mathsf{limit}}{\Gamma, \checkmark_{m'}, \Gamma' \vdash \mathsf{adv}\, t : A} \qquad \frac{\Gamma \vdash t : \Box A}{\Gamma, \sharp, \Gamma' \vdash \mathsf{unbox}\, t : A}$$

$$\frac{\Gamma, \sharp \vdash t : A}{\Gamma \vdash \mathsf{box}\, t : \Box A} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathsf{now}\, t : A\ \mathcal{U}\ B} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : \bigcirc(A\ \mathcal{U}\ B)}{\Gamma \vdash \mathsf{wait}\, s\, t : A\ \mathcal{U}\ B}$$

$$\frac{\Gamma, \sharp, x : B \vdash s : C \qquad \Gamma, \sharp, x : A, y : \bigcirc(A\ \mathcal{U}\ B), z : \bigcirc C \vdash t : C \qquad \Gamma, \sharp, \Gamma' \vdash u : A\ \mathcal{U}\ B}{\Gamma, \sharp, \Gamma' \vdash \mathsf{rec}_{\mathcal{U}}(x.s, x\, y\, z.t, u) : C}$$

$$\frac{\Gamma, x : \Box\triangleright A, \sharp \vdash t : A}{\Gamma \vdash \mathsf{fix}\, x.t : \Box A} \qquad \frac{\Gamma \vdash t : \mathsf{Fix}\, \alpha.A}{\Gamma \vdash \mathsf{out}\, t : A[\triangleright(\mathsf{Fix}\, \alpha.A)/\alpha]} \qquad \frac{\Gamma \vdash t : A[\triangleright(\mathsf{Fix}\, \alpha.A)/\alpha]}{\Gamma \vdash \mathsf{into}\, t : \mathsf{Fix}\, \alpha.A}$$

Fig. 3. Typing rules. Here m, m' ranges over the set $\{\bigcirc, \triangleright\}$ of time modalities ordered by $\bigcirc \leqslant \triangleright$. In all rules, all contexts are assumed well-formed.

states that t has type A one time step after Γ, and so delay t has type $\bigcirc A$ at the time of Γ. Similarly, in the conclusion of the elimination rule, one time step has passed since the premise, so at that time t can be advanced to give an element of type A. This gives a direct approach to programming with modalities, as opposed to the more standard let-expressions. For example, delayed application of functions can be typed as

$$\lambda f.\lambda x.\mathsf{delay}((\mathsf{adv}\, f)(\mathsf{adv}\, x)) : \bigcirc(A \to B) \to \bigcirc A \to \bigcirc B \tag{1}$$

Unlike Simply RaTT, Lively RaTT has two modalities for time delays: $\bigcirc$ and $\triangleright$. Both correspond to a time step in the execution of reactive programs, but in addition, $\triangleright$ corresponds to a time step in the sense of guarded recursion. Consequently, the $\checkmark_\triangleright$ token is stronger than $\checkmark_\bigcirc$: Both can be used to advance time, but $\checkmark_\triangleright$ can also be used to unfold fixed points. We capture this extra strength in a reflexive ordering generated by $\bigcirc \leqslant \triangleright$ on delay modalities, and allowing $\checkmark_{m'}$ to eliminate modality m if $m \leqslant m'$. This induces an inclusion

$$\mathit{embed} = \lambda x.\mathsf{delay}(\mathsf{adv}\, x) : \bigcirc A \to \triangleright A \tag{2}$$

for all A. In general there is no inclusion in the opposite direction, except for a class of special types which we refer to as limit types, defined in Figure 1. The terminology refers to the step indexed interpretation of types, see section 5.

The tokens $\checkmark_{\triangleright}$ and $\checkmark_{\bigcirc}$ are collectively referred to as *ticks* and the rules in Figure 2 stipulate that there may be at most one tick in a context. This means that programs can refer to data from the present and previous time step, but not from earlier time steps. This is a crucial restriction that rules out implicit space leaks, and similar restrictions can be found in many other modal languages for FRP [Bahr et al. 2019; Cave et al. 2014; Krishnaswami 2013].

The second kind of token in Lively RaTT is $\sharp$, which separates the context into static variables to the left of $\sharp$ and dynamic variables to the right. Static variables are time-independent whereas the dynamic ones can depend on reactive data available only in the current instant. This distinction is only made once, so there can be at most one $\sharp$ in a context. The notion of time step is relevant only for dynamic variables, and therefore tokens $\checkmark_{\bigcirc}$ and $\checkmark_{\triangleright}$ can only appear to the right of a $\sharp$. The rules for well-formed contexts can be found in Figure 2.

The token $\sharp$ is associated with the modality $\Box$. Data of type $\Box A$ should be thought of as stable data, i.e., data that does not depend on time-dependent dynamic data, and can thus be safely transported into the future without causing space leaks. This is reflected in the introduction rule for $\Box$ which ensures that $\mathsf{box}\, t$ can not contain free dynamic variables (i.e. variables to the right of $\sharp$), and in the elimination rule allowing $\Gamma \vdash t : \Box A$ to be eliminated in context $\Gamma, \sharp, \Gamma'$ also when Γ' contains a tick.

Stable types (Figure 1) are types whose values by nature cannot contain time-dependent data, and so can be used in any dynamic context. This is implemented in the language by allowing variables of stable types to be introduced also over tokens. Generalising this to all variables would lead to space leaks. Note that function types are not stable since closures can contain time-dependent data.

Guarded recursive types are types of the form $\mathsf{Fix}\ \alpha.A$ satisfying the type isomorphism $\mathsf{Fix}\ \alpha.A \cong A[\triangleright(\mathsf{Fix}\ \alpha.A)/\alpha]$. Note that there is no restriction on A, which can in principle contain also negative occurrences of α, although none of the examples presented in this paper have this. The basic FRP types of streams and events can be encoded as guarded recursive types

$$\mathsf{Str}(A) \stackrel{\text{def}}{=} \mathsf{Fix}\ \alpha.A \times \alpha \qquad\qquad \mathsf{Ev}(A) \stackrel{\text{def}}{=} \mathsf{Fix}\ \alpha.A + \alpha$$

The fixed point combinator as defined by Nakano [2000] is simply a term of type $(\triangleright A \to A) \to A$. In FRP a few adjustments must be made to that. First of all, a fixed point will be called repeatedly at different dynamic times. To avoid space leaks, fixed points should therefore not have free dynamic variables (although the recursion variable itself should be dynamic), and the type of the fixed point should be of the form $\Box A$. In Simply RaTT, the typing rule for fixed points states that $\Gamma \vdash \mathsf{fix}\ x.t : \Box A$ if $\Gamma, \sharp, x : \triangleright A \vdash t : A$. In Lively RaTT this is too restrictive, since it prohibits nesting of guarded fixed points and recursion over elements of the until-types $A\ \mathcal{U}\ B$. The premise of the rule is therefore $\Gamma, x : \Box\triangleright A, \sharp \vdash t : A$, which gives a more general fixed point rule.

As an example, mapping of functions over streams can be defined using fixed points as

$$\begin{aligned} &map : \Box(A \to B) \to \Box(\mathsf{Str}\ A \to \mathsf{Str}\ B)\\ &map = \lambda f\,.\,\mathsf{fix}\ m\,.\,\lambda a :: as\,.\,(\mathsf{unbox}\ f)\ a :: \mathsf{delay}\ ((\mathsf{adv}\ (\mathsf{unbox}\ m))\ (\mathsf{adv}\ as)) \end{aligned}$$

where :: refers to the infix constructor for streams, which in the example is also used for pattern matching. Note that the input function f has type $\Box(A \to B)$ since it must be called at all futures.

Lively RaTT features two kinds of inductive types. The first is the natural numbers with essentially the standard typing rules for 0, suc and recursion. Note that these apply in any context Γ independent of which tokens are in Γ. The second is the until-type of LTL, to be thought of as the inductive solution to $A\ \mathcal{U}\ B \cong B + A \times \bigcirc(A\ \mathcal{U}\ B)$. As for the natural numbers, there is no restriction

on the context for the introduction rules, but the elimination rule is by nature dynamic, since elimination of an element of $A \times \bigcirc(A \mathbin{\mathcal{U}} B)$ should recurse one time step from now on the advanced element of type $A \mathbin{\mathcal{U}} B$. To avoid space leaks, the recursors should be stable, i.e., not depend on dynamic data. Thus eliminating from $A \mathbin{\mathcal{U}} B$ into a type C requires recursors of type $\square(B \to C)$ and $\square(A \to \bigcirc(A \mathbin{\mathcal{U}} B) \to \bigcirc C \to C)$.

Finally, note that Lively RaTT is a higher-order functional programming language with the restriction that lambda abstraction is only allowed in contexts with no $\surd_{\bigcirc}$ and $\surd_{\triangleright}$. This restriction is inherited from Simply RaTT where it is necessary to guarantee the lack of space leaks. As we shall see, this appears not to be a limitation in practice.

3 PROGRAMMING IN LIVELY RATT

This section gives a number of examples of programming in Lively RaTT. First, we give a series of examples of programming with events. Secondly, we show how to encode *fairness* and how to implement a fair scheduler.

3.1 Events and Diamond

As described in the introduction, events that *may* occur can be encoded in Lively RaTT as $\mathsf{Ev}\, A \stackrel{\text{def}}{=} \mathsf{Fix}\, \alpha.A + \alpha$. For example, the event that loops forever can be defined as

$$
\begin{aligned}
&\mathit{loopEvent} : \square \mathsf{Ev}\ A \\
&\mathit{loopEvent} = \mathsf{fix}\ e\,.\,\mathsf{into}\ (\mathsf{in}_2\ (\mathsf{unbox}\ e))
\end{aligned}
$$

The type of events that *must* occur can be encoded as the diamond modality from LTL, namely $\Diamond(A) \stackrel{\text{def}}{=} 1 \mathbin{\mathcal{U}} A$. Below we will use the following shorthand when working with Ev and $\Diamond$:

$$
\begin{aligned}
&\mathsf{now}_{\Diamond} : A \to \Diamond A && \mathsf{now}_{\mathsf{Ev}} : A \to \mathsf{Ev}\ A \\
&\mathsf{now}_{\Diamond}\ a = \mathsf{now}\ a && \mathsf{now}_{\mathsf{Ev}}\ a = \mathsf{into}\ (\mathsf{in}_1\ a) \\
&\mathsf{wait}_{\Diamond} : \bigcirc\Diamond A \to \Diamond A && \mathsf{wait}_{\mathsf{Ev}} : \triangleright \mathsf{Ev}\ A \to \mathsf{Ev}\ A \\
&\mathsf{wait}_{\Diamond}\ d = \mathsf{wait}\ \langle\rangle\ d && \mathsf{wait}_{\mathsf{Ev}}\ e = \mathsf{into}\ (\mathsf{in}_2\ e)
\end{aligned}
$$

Here the $\mathsf{now}_{\Diamond}$ and $\mathsf{now}_{\mathsf{Ev}}$ maps are like the return map from a monad. Both Ev and $\Diamond$ further admit a map reminiscent of the bind map of a monad. For Ev this is given by:

$$
\begin{aligned}
&\mathit{bind}_{\mathsf{Ev}} : \square(A \to \mathsf{Ev}\ B) \to \square(\mathsf{Ev}\ A \to \mathsf{Ev}\ B) \\
&\mathit{bind}_{\mathsf{Ev}} = \lambda f\,.\,(\mathsf{fix}\ b\,.\,\lambda eva\,.\,\mathrm{case}\ eva\ \mathrm{of}\ \mathsf{now}_{\mathsf{Ev}}\ a\,.\,(\mathsf{unbox}\ f)\ a \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{wait}_{\mathsf{Ev}}\ e\,.\,\mathsf{wait}_{\mathsf{Ev}}\ (\mathsf{unbox}\ b \circledast e))
\end{aligned}
$$

where $\circledast$ is the infix notation of the delayed function call as defined in Equation 1, which can be given the more general type $m_1(A \to B) \to m_2(A) \to m_3(B)$ for $m_i \in \{\bigcirc, \triangleright\}$ with $m_1, m_2 \leqslant m_3$. To see that $\mathit{bind}_{\mathsf{Ev}}$ is well-typed, consider the two cases. In the first case $a : A$ and hence, the unboxed f can be applied immediately. In the second case $e : \triangleright \mathsf{Ev}\, A$ and $b : \square\triangleright(\mathsf{Ev}\, A \to \mathsf{Ev}\, B)$. It then follows that $\mathsf{unbox}\, b : \triangleright(\mathsf{Ev}\, A \to \mathsf{Ev}\, B)$ and thus, by a delayed function application, $(\mathsf{unbox}\, b) \circledast e : \triangleright \mathsf{Ev}\, B$. This is then wrapped in $\mathsf{wait}_{\mathsf{Ev}}$ to produce an element of $\mathsf{Ev}\, B$ as needed. Note the requirement for the map f to be stable. This is because it might be applied *in the future*.

For $\Diamond$, we define the map

$$
\begin{aligned}
&\mathit{bind}_{\Diamond} : \square(A \to \Diamond B) \to \square(\Diamond A \to \Diamond B) \\
&\mathit{bind}_{\Diamond} = \lambda f\,.\,\mathsf{box}\ (\lambda \mathit{dia}\,.\,\mathsf{rec}_{\mathcal{U}}\ (a\,.\,(\mathsf{unbox}\ f)\ a, u\ w\ d\,.\,\mathsf{wait}_{\Diamond}\ d, \mathit{dia}))
\end{aligned}
$$

where again f must be stable. To see that $\mathit{bind}_{\Diamond}$ is well typed, consider the base and recursion case. In the base $a : A$ and hence, the unboxed f can be applied immediately. In the recursion case $u : 1, w : \bigcirc(A \mathbin{\mathcal{U}} B)$ and $d : \bigcirc\Diamond B$, hence also $\mathsf{wait}_{\Diamond}\ d : \Diamond B$ as required.

We will also use sugared syntax for recursive definitions, writing e.g. the definition of $bind_{\mathsf{Ev}}$ as

$$
\begin{aligned}
&bind_{\mathsf{Ev}} : \Box(A \to \mathsf{Ev}\ B) \to \Box(\mathsf{Ev}\ A \to \mathsf{Ev}\ B)\\
&bind_{\mathsf{Ev}}\ f\ \sharp\ (\mathsf{now}_{\mathsf{Ev}}\ a) = (\mathsf{unbox}\ f)\ a\\
&bind_{\mathsf{Ev}}\ f\ \sharp\ (\mathsf{wait}_{\mathsf{Ev}}\ e) = \mathsf{wait}_{\mathsf{Ev}}\ (\mathsf{unbox}\ (bind_{\mathsf{Ev}}\ f) \circledast e)
\end{aligned}
$$

The $\sharp$ separates the variables into those received before and after fix, and since the two cases define $bind_{\mathsf{Ev}}\ f$ by guarded recursion, this should be considered an atomic subexpression with type $\Box\triangleright(\mathsf{Ev}\,A \to \mathsf{Ev}\,B)$. Similarly, the definition of $bind_{\Diamond}$ can be written in the sugared syntax as

$$
\begin{aligned}
&bind_{\Diamond} : \Box(A \to \Diamond B) \to \Box(\Diamond A \to \Diamond B)\\
&bind_{\Diamond}\ f = \mathsf{box}\ bind'_{\Diamond}\\
&\text{where } bind'_{\Diamond} : \Diamond A \to \Diamond B\\
&\qquad bind'_{\Diamond}\ (\mathsf{now}_{\Diamond}\ a) = (\mathsf{unbox}\ f)\ a\\
&\qquad bind'_{\Diamond}\ (\mathsf{wait}_{\Diamond}\ d) = \mathsf{wait}_{\Diamond}\ (bind'_{\Diamond}\ d)
\end{aligned}
$$

Here, the two cases of the recursive definition of $bind'_{\Diamond}$ are written as pattern matching syntax. In the second case the subterm $bind'_{\Diamond}\ d$ represents the recursive call and should therefore be read as having type $\bigcirc\Diamond B$. To elaborate such definitions back into $\mathsf{rec}_{\mathcal{U}}$, replace calls such as $bind'_{\Diamond}\ d$ with a fresh variable that represents the call to the recursor. We chose to use the above style to make it clear when delayed arguments are used and how they are passed around. See Appendix A for an overview of the elaboration process from surface syntax to the core calculus.

Since $\Diamond$ represents events that must occur, and Ev represents more general, possibly occurring, events there is an inclusion from $\Diamond$ to Ev. Using the above syntax, this is by $\mathcal{U}$-recursion as

$$
\begin{aligned}
&diaInclusion : \Box(\Diamond A \to \mathsf{Ev}\ A)\\
&diaInclusion = \mathsf{box}\ diaInclusion'\\
&\text{where } diaInclusion' : \Diamond A \to \mathsf{Ev}\ A\\
&\qquad diaInclusion'\ (\mathsf{now}_{\Diamond}\ a) = \mathsf{now}_{\mathsf{Ev}}\ a\\
&\qquad diaInclusion'\ (\mathsf{wait}_{\Diamond}\ d) = \mathsf{wait}_{\mathsf{Ev}}\ (embed\ (diaInclusion'\ d))
\end{aligned}
$$

This map makes crucial use of the fact that $\bigcirc$ is a sub-modality of $\triangleright$ in the call to $embed$, as defined in Equation 2.

A further consequence of the sub-modality relation is that non-terminating events "overrule" terminating events. Consider Ev containing a $\Diamond$:

$$
\begin{aligned}
&diamondEvent : \Box(\mathsf{Ev}\Diamond A \to \mathsf{Ev}\ A)\\
&diamondEvent = bind_{\mathsf{Ev}}\ diaInclusion
\end{aligned}
$$

The converse, a function with type $\Diamond\mathsf{Ev}\,A \to \Diamond\,A$, can not be written in the language, since the inner event may be non-terminating.

There is in general no inclusion from Ev into $\Diamond$ because of the requirement that elements of $\Diamond A$ terminate. One solution is to wrap the conversion in a timeout, which will handle the non-terminating case. We must then supply a natural number, representing how many time steps to wait, and let the conversion fail if we go beyond that. We define by natural number recursion

$$
\begin{aligned}
&timeout : \mathsf{limit}\ A \Rightarrow \mathsf{Nat} \to \Box(\mathsf{Ev}\ A \to \Diamond(1 + A))\\
&timeout\ 0\qquad\ \ \sharp\ (\mathsf{now}_{\mathsf{Ev}}\ a) = \mathsf{now}_{\Diamond}\ (\mathsf{in}_2\ a)\\
&timeout\ 0\qquad\ \ \sharp\ (\mathsf{wait}_{\mathsf{Ev}}\ e) = \mathsf{now}_{\Diamond}\ (\mathsf{in}_1\ \langle\rangle))\\
&timeout\ (\mathsf{suc}\ n)\ \sharp\ (\mathsf{now}_{\mathsf{Ev}}\ a) = \mathsf{now}_{\Diamond}\ (\mathsf{in}_2\ a)\\
&timeout\ (\mathsf{suc}\ n)\ \sharp\ (\mathsf{wait}_{\mathsf{Ev}}\ e) = \mathsf{wait}_{\Diamond}\ (\mathsf{delay}\ ((\mathsf{unbox}\ (timeout\ n))\ (\mathsf{adv}\ e)))
\end{aligned}
$$

Typing of the fourth case uses the requirement $A\,\mathsf{limit}$: The delay corresponds to a $\bigcirc$-tick in the

context, which in general can not be used to advance $e : \triangleright \mathsf{Ev}\, A$. It can in this case since A limit implies $\triangleright \mathsf{Ev}\, A$ limit.

Even though we can not give a general function $\Diamond \mathsf{Ev} A \to \Diamond A$, we can use the above $timeout$ to give a function $\Diamond \mathsf{Ev} A \to \Diamond(1 + A)$.

$$
\begin{aligned}
&eventDiamond : \mathsf{limit}\ A \Rightarrow \mathsf{Nat} \to \Box(\Diamond \mathsf{Ev}\ A \to \Diamond(1 + A)) \\
&eventDiamond\ n = bind_{\Diamond}\ (timeout\ n)
\end{aligned}
$$

In general we cannot join two events of type $\Diamond A$ and $\Diamond B$ to an event of type $\Diamond(A \times B)$, i.e., waiting for both events to occur and pairing up the result. Doing so for arbitrary types A and B may lead to space leaks: If the two events occur at different times, the result of whichever event occurs first would need to be buffered until the second one occurs. However, if A and B are stable, we can explicitly buffer the early event occurrence, which allows us to implement the join. The implementation relies on two auxiliary functions that differ only in which order their arguments are given. We give only one of them, implemented by $\mathcal{U}$-recursion:

$$
\begin{aligned}
&joinAuxA : A\ \mathsf{stable} \Rightarrow \Box(\Diamond B \to A \to \Diamond(A \times B)) \\
&joinAuxA = \mathsf{box}\ joinAuxA' \\
&\text{where } joinAuxA'\ (\mathsf{now}_{\Diamond}\ b)\ a\ = \mathsf{now}_{\Diamond}\ \langle a, b \rangle \\
&\qquad joinAuxA'\ (\mathsf{wait}_{\Diamond}\ d)\ a = \mathsf{wait}_{\Diamond}\ ((joinAuxA'\ d) \odot a)
\end{aligned}
$$

where $\odot$ is the infix function

$$
\begin{aligned}
&\_ \odot \_ : A\ \mathsf{stable} \Rightarrow m_1\ (A \to B) \to A \to m_2\ B \\
&f \odot a = \mathsf{delay}\ ((\mathsf{adv}\ f)\ a)
\end{aligned}
$$

where $m_1, m_2 \in \{\bigcirc, \triangleright\}$ and $m_1 \leqslant m_2$. With the above, we can now define the join by $\mathcal{U}$-recursion:

$$
\begin{aligned}
&join : A, B\ \mathsf{stable} \Rightarrow \Box(\Diamond A \to \Diamond B \to \Diamond(A \times B)) \\
&join = \mathsf{box}\ join' \\
&\text{where } join' : \Diamond A \to \Diamond B \to \Diamond(A \times B) \\
&\qquad join'\ (\mathsf{now}_{\Diamond}\ a)\ (\mathsf{now}_{\Diamond}\ b)\ = \mathsf{now}_{\Diamond}\ (a, b) \\
&\qquad join'\ (\mathsf{now}_{\Diamond}\ a)\ (\mathsf{wait}_{\Diamond}\ d)\ = (\mathsf{unbox}\ joinAuxA)\ (\mathsf{wait}_{\Diamond}\ d)\ a \\
&\qquad join'\ (\mathsf{wait}_{\Diamond}\ d)\ (\mathsf{now}_{\Diamond}\ b)\ = (\mathsf{unbox}\ joinAuxB)\ (\mathsf{wait}_{\Diamond}\ d)\ b \\
&\qquad join'\ (\mathsf{wait}_{\Diamond}\ d)\ (\mathsf{wait}_{\Diamond}\ d') = \mathsf{wait}_{\Diamond}\ ((join'\ d) \circledast d')
\end{aligned}
$$

We now give a function constructing elements of $\Diamond$ by buffering data for a given number of time steps. For this we need a type of *temporal natural numbers* that will serve as a means to count time:

$$\mathsf{Nat}_{\bigcirc} = \Diamond 1$$

This type can be thought of as natural numbers, where the successor operation requires one time step to compute. The zero and successor can be encoded as:

$$
\begin{aligned}
0_{\bigcirc} &= \mathsf{now}_{\Diamond} \langle\rangle \\
\mathsf{suc}_{\bigcirc}\ n &= \mathsf{wait}_{\Diamond}\ n
\end{aligned}
$$

Any temporal natural number can be imported into the future by means of $\mathcal{U}$-recursion.

$$
\begin{aligned}
&import : \mathsf{Nat}_{\bigcirc} \to \bigcirc \mathsf{Nat}_{\bigcirc} \\
&import\ 0_{\bigcirc} \qquad\ = \mathsf{delay}\ (0_{\bigcirc}) \\
&import\ (\mathsf{suc}_{\bigcirc}\ n) = \mathsf{delay}\ (\mathsf{suc}_{\bigcirc}\ (\mathsf{adv}\ (import\ n)))
\end{aligned}
$$

Given a natural number, we can convert it into a temporal natural number by recursion on natural numbers:

$$
\begin{aligned}
&\mathit{timer} : \mathsf{Nat} \to \mathsf{Nat}_{\bigcirc} \\
&\mathit{timer}\ 0 \qquad\ = 0_{\bigcirc} \\
&\mathit{timer}\ (\mathsf{suc}\ n) = \mathsf{suc}_{\bigcirc}\ (\mathit{import}\ (\mathit{timer}\ n))
\end{aligned}
$$

Intuitively speaking, given a natural number n, $\mathit{timer}\ n$ is a timer with n ticks.

The buffer function takes a temporal natural number and requires A to be stable, for the input to be buffered.

$$
\begin{aligned}
&\mathit{buffer} : A\ \mathsf{stable} \Rightarrow \Box(\mathsf{Nat}_{\bigcirc} \to A \to \Diamond A) \\
&\mathit{buffer} = \mathsf{box}\ \mathit{buffer}' \\
&\text{where } \mathit{buffer}' : \mathsf{Nat}_{\bigcirc} \to A \to \Diamond A \\
&\qquad \mathit{buffer}'\ 0_{\bigcirc}\ a \qquad\ = \mathsf{now}_{\Diamond}\ a \\
&\qquad \mathit{buffer}'\ (\mathsf{suc}_{\bigcirc}\ n)\ a = \mathsf{wait}_{\Diamond}\ ((\mathit{buffer}'\ n) \odot a)
\end{aligned}
$$

As a final example of working with $\Diamond$ we define a simple server. First off we define the type of servers as

$$\mathsf{Server} := \mathsf{Fix}\,\alpha.\alpha \times (\mathsf{Req} \to (\Diamond\mathsf{Resp} \times \alpha))$$

where Req and Resp are the types of requests and responses, respectively. In each step, a server can receive at most one request, which must eventually give a response. In either case, the server will return a new server in the next time step, with a possibly updated internal state.

With the above, we can define a simple server that given a string s and a number n, returns $\langle s, m\rangle$ after n time steps, where m is the number of requests received. We set $\mathsf{Req} := \mathsf{Nat} \times \mathsf{String}$ and $\mathsf{Resp} := \mathsf{String} \times \mathsf{Nat}$, and consider String to be stable. The server is defined by guarded recursion:

$$
\begin{aligned}
&\mathit{rServer} : \Box(\mathsf{Nat} \to \mathsf{Server}) \\
&\mathit{rServer}\ \sharp\ m = \mathsf{into}\ \langle \mathit{rServerFst}, \mathit{rServerSnd}\rangle \\
&\text{where } \mathit{rServerFst} : \triangleright\mathsf{Server} \\
&\qquad \mathit{rServerFst} = (\mathsf{unbox}\ \mathit{rServer}) \odot m \\
&\qquad \mathit{rServerSnd} : (\mathsf{Nat} \times \mathsf{String}) \to (\Diamond(\mathsf{String} \times \mathsf{Nat}) \times \triangleright\mathsf{Server}) \\
&\qquad \mathit{rServerSnd}\ \langle n, s\rangle = \langle (\mathsf{unbox}\ \mathit{buffer})\ (\mathit{timer}\ n)\ \langle s, m\rangle, (\mathsf{unbox}\ \mathit{rServer}) \odot (\mathsf{suc}\ m)\rangle
\end{aligned}
$$

The server can be run and initialized with 0:

$$
\begin{aligned}
&\mathit{rServerRun} : \Box\mathsf{Server} \\
&\mathit{rServerRun} = \mathsf{box}\ ((\mathsf{unbox}\ \mathit{rServer})\ 0)
\end{aligned}
$$

3.2 Fair Streams

A stream of type $\mathsf{Str}(A + B)$ will in each step produce either a value of type A or of B. For example, we can implement a scheduler that interleaves two streams in an alternating fashion, dropping every other element of either stream:

$$
\begin{aligned}
&\mathit{altStr} : \Box(\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Str}\ (A + B)) \\
&\mathit{altStr}\ \sharp\ (a :: as)\ (b :: bs) = \mathsf{in}_2\ b :: (\mathsf{unbox}\ \mathit{altStr}' \circledast as \circledast bs)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{altStr}' : \Box(\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Str}\ (A + B)) \\
&\mathit{altStr}'\ \sharp\ (a :: as)\ (b :: bs) = \mathsf{in}_1\ a :: (\mathsf{unbox}\ \mathit{altStr} \circledast as \circledast bs)
\end{aligned}
$$

altStr and altStr' are defined by mutual guarded recursion. The former starts with taking an element from the second stream whereas the latter starts with taking from the first stream. This mutual recursive syntax translates to a single guarded fixed point in the calculus via a standard tupling construction that constructs both functions simultaneously:

$$altStrMut : \Box((\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Str}\ (A + B)) \times (\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Str}\ (A + B)))$$
$$altStrMut = \mathsf{fix}\ r\,.\,\langle \lambda as\,.\,\lambda bs\,.\,\mathsf{in}_2\ (\mathsf{head}\ bs) :: (\pi_2^{\triangleright}\ (\mathsf{unbox}\ r) \circledast \mathsf{tail}\ as \circledast \mathsf{tail}\ bs),$$
$$\lambda as\,.\,\lambda bs\,.\,\mathsf{in}_1\ (\mathsf{head}\ as) :: (\pi_1^{\triangleright}\ (\mathsf{unbox}\ r) \circledast \mathsf{tail}\ as \circledast \mathsf{tail}\ bs)\rangle$$

In the above definition, the variable r is of type

$$\Box\triangleright((\mathsf{Str}(A) \to \mathsf{Str}(B) \to \mathsf{Str}(A + B)) \times (\mathsf{Str}(A) \to \mathsf{Str}(B) \to \mathsf{Str}(A + B)))$$

We then use the projections π_i lifted to $\triangleright$ to access the desired component from $\mathsf{unbox}\, r$:

$$\pi_i^{\triangleright} : \triangleright(A_1 \times A_2) \to \triangleright A_i$$
$$\pi_i^{\triangleright} = \lambda x.\mathsf{delay}(\pi_i(\mathsf{adv}\, x))$$

Given the above tupling construction, $altStr$ is then defined as $\mathsf{box}(\pi_1\ (\mathsf{unbox}\ altStrMut))$. From now on we will use the mutual guarded recursion syntax with the understanding that it can be turned into a single guarded fixed point by tupling.

The following function inhabits the same type as $altStr$, but it only draws elements from the first stream, dropping the second stream altogether:

$$dropSnd : \Box(\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Str}\ (A + B))$$
$$dropSnd\ \sharp\ as\ bs = \mathsf{unbox}\ (map\ (\mathsf{box}\ \mathsf{in}_1))\ as$$

Following the work by Cave et al. [2014], we can refine the type $\mathsf{Str}(A + B)$ to a type $\mathsf{Fair}(\mathsf{A}, \mathsf{B})$, whose inhabitants will produce in each step a value of type A or of type B, in a fair manner:

$$\mathsf{Fair}(\mathsf{A}, \mathsf{B}) = \mathsf{Fix}\,\alpha.A\ \mathcal{U}\ (B \times \triangleright(B\ \mathcal{U}\ (A \times \alpha)))$$

A term of type $\mathsf{Fair}(\mathsf{A}, \mathsf{B})$ may first produce some elements of type A, but must after finitely many steps produce an element of type B. It may continue to produce more elements of type B, but must eventually produce an element of type A and then continue in this manner indefinitely. This required behaviour prevents us from implementing $dropSnd$ to produce a fair stream of type $\mathsf{Fair}(\mathsf{A}, \mathsf{B})$. On the other hand, we can re-implement $altStr$ to produce a fair stream as follows:

$$altFair\ : \Box(\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Fair}(A, B))$$
$$altFair\ \sharp\ (a :: as)\ (b :: bs) = \mathsf{into}\ (\mathsf{now}\ \langle b, \mathsf{unbox}\ altFair' \circledast as \circledast bs\rangle)$$
$$altFair' : \Box(\mathsf{Str}\ A \to \mathsf{Str}\ B \to B\ \mathcal{U}\ (A \times \triangleright\mathsf{Fair}(A, B)))$$
$$altFair'\ \sharp\ (a :: as)\ (b :: bs) = (\mathsf{now}\ \langle a, \mathsf{unbox}\ altFair \circledast as \circledast bs\rangle)$$

In order to simplify programming with fair streams, we define shortcut constructors for the type $\mathsf{Fair}(\mathsf{A}, \mathsf{B})$. To this end we define the following variant of the type $\mathsf{Fair}(\mathsf{A}, \mathsf{B})$:

$$\mathsf{Fair}'(\mathsf{B}, \mathsf{A}) = B\ \mathcal{U}\ (A \times \triangleright\mathsf{Fair}(\mathsf{A}, \mathsf{B}))$$

We now have that $\mathsf{Fair}(\mathsf{A}, \mathsf{B})$ unfolds to $A\ \mathcal{U}\ (B \times \triangleright\mathsf{Fair}'(\mathsf{B}, \mathsf{A}))$ and thus the two types $\mathsf{Fair}(\mathsf{A}, \mathsf{B})$ and $\mathsf{Fair}'(\mathsf{A}, \mathsf{B})$ are isomorphic. Fair streams are constructed by either staying with the first type A or switching to the second type B.

$$\mathsf{stay} : A \to \bigcirc\mathsf{Fair}(A, B) \to \mathsf{Fair}(A, B)$$
$$\mathsf{stay}\ a\ d\ \ = \mathsf{into}\ (\mathsf{wait}\ a\ d)$$
$$\mathsf{stay}' : A \to \bigcirc\mathsf{Fair}'(A, B) \to \mathsf{Fair}'(A, B)$$
$$\mathsf{stay}'\ a\ d\ \ = \mathsf{wait}\ a\ d$$
$$\mathsf{switch} : B \to \triangleright\mathsf{Fair}'(B, A) \to \mathsf{Fair}(A, B)$$
$$\mathsf{switch}\ b\ d = \mathsf{into}\ (\mathsf{now}\ \langle b, d\rangle)$$
$$\mathsf{switch}' : B \to \triangleright\mathsf{Fair}(B, A) \to \mathsf{Fair}'(A, B)$$
$$\mathsf{switch}'\ b\ d = \mathsf{now}\ \langle b, d\rangle$$

From the types above one can immediately see that we can only stay with the same type finitely often – indicated by the $\bigcirc$ modality – whereas we can switch arbitrarily – indicated by the $\triangleright$ modality. Using the above shorthands, the $altFair$ function can thus be implemented more concisely as follows:

$$
\begin{aligned}
&\mathit{altFair} \;: \Box(\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Fair}(A,B))\\
&\mathit{altFair}\ \sharp\ (a :: as)\ (b :: bs) = \mathsf{switch}\ b\ (\mathsf{unbox}\ \mathit{altFair}' \circledast as \circledast bs)\\
&\mathit{altFair}' : \Box(\mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Fair}'(B,A))\\
&\mathit{altFair}'\ \sharp\ (a :: as)\ (b :: bs) = \mathsf{switch}'\ a\ (\mathsf{unbox}\ \mathit{altFair} \circledast as \circledast bs)
\end{aligned}
$$

The fair stream type Fair(A, B) can be considered a special case of the stream type $\mathsf{Str}(A + B)$ with additional liveness constraints. We can always forget these constraints by converting a fair stream into a normal stream:

$$
\begin{aligned}
&\mathit{runFair} : \Box(\mathsf{Fair}(A,B) \to \mathsf{Str}\ (A + B))\\
&\mathit{runFair}\ \sharp\ = \mathit{run}_1\\
&\quad \text{where } \mathit{run}_2 : \mathit{Fair}'\ (B,A) \to \mathsf{Str}\ (A + B)\\
&\qquad\quad \mathit{run}_2\ (\mathsf{stay}'\ b\ d) \quad\ = \mathsf{in}_2\ b :: \mathit{embed}\ (\mathit{run}_2\ d)\\
&\qquad\quad \mathit{run}_2\ (\mathsf{switch}'\ a\ d) = \mathsf{in}_1\ a :: \mathsf{unbox}\ \mathit{runFair} \circledast d\\
&\qquad\quad \mathit{run}_1 : \mathsf{Fair}\ (A,B) \to \mathsf{Str}\ (A + B)\\
&\qquad\quad \mathit{run}_1\ (\mathsf{stay}\ a\ d) \quad\ = \mathsf{in}_1\ a :: \mathit{embed}\ (\mathit{run}_1\ d)\\
&\qquad\quad \mathit{run}_1\ (\mathsf{switch}\ b\ d) = \mathsf{in}_2\ b :: \mathsf{delay}\ (\mathit{run}_2\ (\mathsf{adv}\ d))
\end{aligned}
$$

The function $\mathit{runFair}$ is defined by guarded recursion with two nested $\mathcal{U}$-recursions on the two nested $\mathcal{U}$-types that make up the fair stream type. Note that the two recursive calls $\mathit{run}_1\, d$ and $\mathit{run}_2\, d$ produce a delayed stream of type $\bigcirc(\mathsf{Str}(A + B))$. Therefore, we have to use embed to convert them to type $\triangleright(\mathsf{Str}(A + B))$.

We conclude with an example that implements a more interesting interleaving of two streams into a fair stream, namely the fair scheduler from Cave et al. [2014] that selects a progressively increasing number of elements from the first stream for each time it selects an element from the second stream:

$$
\begin{aligned}
&\mathit{sch}\ : \mathsf{limit}\ A, \mathsf{limit}\ B \Rightarrow \Box(\mathsf{Nat} \to \mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Fair}(A,B))\\
&\mathit{sch}\ \sharp\ n\ as\ bs = \mathit{until}\ (\mathit{timer}\ n)\ n\ as\ bs\\
&\quad \text{where } \mathit{until} : \mathsf{Nat}_\bigcirc \to \mathsf{Nat} \to \mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Fair}(A,B))\\
&\qquad\quad \mathit{until}\ (\mathsf{suc}_\bigcirc\ n)\ m\ (a :: as)\ (b :: bs) = \mathsf{stay}\ a\ (\mathit{until}\ n \odot m \circledast as \circledast bs))\\
&\qquad\quad \mathit{until}\ 0_\bigcirc\ m\ (a :: as)\ (b :: bs)\\
&\qquad\qquad = \mathsf{switch}\ b\ (\mathsf{unbox}\ \mathit{sch}' \odot m + 1 \circledast as \circledast bs)\\
&\mathit{sch}' : \mathsf{limit}\ A, \mathsf{limit}\ B \Rightarrow \Box(\mathsf{Nat} \to \mathsf{Str}\ A \to \mathsf{Str}\ B \to \mathsf{Fair}'(B,A))\\
&\mathit{sch}'\ \sharp\ n\ (a :: as)\ (b :: bs) = \mathsf{switch}'\ a\ (\mathsf{unbox}\ \mathit{sch} \odot n \circledast as \circledast bs)
\end{aligned}
$$

In particular unbox $\mathit{sch}\ 0\ as\ bs$ produces a fair stream of the following form:

$$B\quad A\quad A\quad B\quad A\quad A\quad A\quad B\quad A\quad A\quad A\quad A\quad B\quad A\quad A\quad A\quad A\quad A\quad B\ldots$$

The fair scheduler is implemented by guarded mutual recursion with a nested $\mathcal{U}$-recursion. The natural number is first turned into a timer, which is then recursed over using the until function. In each recursive step of until – corresponding to a tick of the timer – we select from the first stream. But once the timer reaches $0_\bigcirc$, we switch to selecting from the second stream, then immediately switch to selecting from the first stream again, increment the counter *m*, and proceed by guarded recursion.

Note that we require A and B to be limit types so that in turn $\mathsf{Str}(A)$ and $\mathsf{Str}(B)$ are limit types. The latter is needed in the first clause of the until function so that we may apply the recursive call $\mathit{until}\ n$ of type $\bigcirc(\mathsf{Nat} \to \mathsf{Str}(A) \to \mathsf{Str}(B) \to \mathsf{Fair(A, B)})$ to both *as* and *bs*, which are of type $\triangleright\mathsf{Str}(A)$ and $\triangleright\mathsf{Str}(B)$, respectively.

4 OPERATIONAL SEMANTICS

The operational semantics of Lively RaTT is divided into two parts: an *evaluation semantics* that captures the computational behaviour at each time instant (section 4.1), and a *step semantics* that describes the dynamic behaviour of a Lively RaTT program over time. We introduce the latter in two stages. At first we only look at programs without external input (section 4.2). Afterwards we extend the semantics to account for programs that react to external inputs (section 4.3), e.g., terms of type $\Box(\mathsf{Str}(A) \to \mathsf{Str}(B))$, which continuously read inputs of type A and produce outputs of type B. Along the way we give a precise account of our main technical results, namely productivity, termination, liveness, and causality properties of the operational semantics, as well as the absence of implicit space leaks. To prove the latter, the evaluation semantics is formulated using a store on which external inputs and delayed computations are placed. At each reduction step, the step semantics garbage collects all elements of the store that are more than one step old, thereby avoiding implicit space leaks by construction.

4.1 Evaluation Semantics

The *evaluation semantics* is presented as a big-step operational semantics in Figure 4 and describes how a configuration consisting of a term t and a store σ evaluates to a value v and an updated store τ in the current time instant, denoted $\langle t; \sigma\rangle \Downarrow \langle v; \tau\rangle$. In the machine, unlike the surface language, terms can contain locations l, to be thought of as locations in the store. Formally, these range over a given set Loc of locations divided into a countably infinite collection of namespaces each consisting of countably infinitely many locations. The grammar below describes which terms of the calculus are considered values:

$$v, w ::= \langle\rangle \mid 0 \mid \mathsf{suc}\, v \mid \lambda x.t \mid \langle v, w\rangle \mid \mathsf{in}_i\, v \mid \ \mathsf{box}\, t \mid \mathsf{delay}\, t \mid \mathsf{fix}\, x.t \mid l \mid \mathsf{into}\, v \ \mid \mathsf{now}\, v \mid \mathsf{wait}\, v\, w$$

A store can be of one of three forms: $\sigma ::= \bullet \mid \eta_L \mid \eta_N \checkmark \eta_L$. The null store $\bullet$ is used for a special state of the machine in which it can neither write to the store, nor read from it. In the other two forms η_L (the 'later' heap) and η_N (the 'now' heap) are heaps, i.e., pairs of a namespace and a finite mapping from the namespace to terms. In either of the two latter cases, the evaluation semantics can update all heaps present by allocating fresh locations and placing delayed computations in them, but it can only read from the η_N heap. The notation $\sigma, l \mapsto t$ refers to an extension of the heap furthest to the right in σ, and $\mathsf{alloc}\,(-)$ is a function returning a fresh location in the heap furthest to the right.

The fragment of Lively RaTT consisting of the lambda calculus with sums, products, and natural numbers is given a standard call-by-value semantics. The non-standard parts of the semantics involve the three modalities $\Box$, $\bigcirc$, and $\triangleright$; the recursion principle for $\mathcal{U}$ types; and the fixed point combinator.

The constructors for the three modalities – box and delay – have a call-by-name semantics and produce suspended computations. Terms of the form $\mathsf{box}\, t$ are values consisting of unevaluated terms t, which are only evaluated once they are consumed by unbox. Terms of the form $\mathsf{delay}\, t$ are computations that may be executed in the next time step. These computations are suspended and placed on the heap, since the evaluation semantics only describes computations at the current time instant. In the next time instant, these will be executed if adv is applied to their location. The safety of the step semantics shows that such delayed computations will only be executed in the next time step, and do not need to be stored for future steps. Note that in some special cases, a term $\mathsf{delay}(t)$ will be executed immediately, for example when evaluating a closed term of the form $\mathsf{adv}(\mathsf{delay}(t))$. Although such a term is not well-typed as a closed term by itself, it can occur in the evaluation of the step semantics.

Call-by-value λ-calculus:

$$\frac{}{\langle v;\sigma\rangle \Downarrow \langle v;\sigma\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle \qquad \langle t';\sigma'\rangle \Downarrow \langle v';\sigma''\rangle}{\langle\langle t,t'\rangle;\sigma\rangle \Downarrow \langle\langle v,v'\rangle;\sigma''\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow \langle\langle v_1,v_2\rangle;\sigma'\rangle \qquad i\in\{1,2\}}{\langle \pi_i(t);\sigma\rangle \Downarrow \langle v_i;\sigma'\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle \qquad i\in\{1,2\}}{\langle \mathsf{in}_i(t);\sigma\rangle \Downarrow \langle \mathsf{in}_i(v);\sigma'\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow \langle \mathsf{in}_i(v);\sigma'\rangle \qquad \langle t_i[v/x];\sigma'\rangle \Downarrow \langle v_i;\sigma''\rangle \qquad i\in\{1,2\}}{\langle \mathsf{case}\, t\, \mathsf{of}\, \mathsf{in}_1\, x.t_1; \mathsf{in}_2\, x.t_2;\sigma\rangle \Downarrow \langle v_i;\sigma''\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow \langle \lambda x.s;\sigma'\rangle \qquad \langle t';\sigma'\rangle \Downarrow \langle v;\sigma''\rangle \qquad \langle s[v/x];\sigma''\rangle \Downarrow \langle v';\sigma'''\rangle}{\langle t\, t';\sigma\rangle \Downarrow \langle v';\sigma'''\rangle}$$

Modalities, $\mathcal{U}$-types, guarded recursion:

$$\frac{l = \mathsf{alloc}\,(\sigma) \qquad \sigma \neq \bullet}{\langle \mathsf{delay}\, t;\sigma\rangle \Downarrow \langle l;(\sigma, l\mapsto t)\rangle} \qquad \frac{\langle t;\eta_N\rangle \Downarrow \langle l;\eta'_N\rangle \qquad \langle \eta'_N(l);(\eta'_N \checkmark \eta_L)\rangle \Downarrow \langle v;\sigma'\rangle}{\langle \mathsf{adv}\, t;(\eta_N\checkmark\eta_L)\rangle \Downarrow \langle v;\sigma'\rangle}$$

$$\frac{\langle t;\bullet\rangle \Downarrow \langle \mathsf{box}\, t';\bullet\rangle \qquad \langle t';\sigma\rangle \Downarrow \langle v;\sigma'\rangle \qquad \sigma\neq\bullet}{\langle \mathsf{unbox}\, t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}{\langle \mathsf{suc}\, t;\sigma\rangle \Downarrow \langle \mathsf{suc}\, v;\sigma'\rangle}$$

$$\frac{\langle n;\sigma\rangle \Downarrow \langle 0;\sigma'\rangle \qquad \langle s;\sigma'\rangle \Downarrow \langle v;\sigma''\rangle}{\langle \mathsf{rec}_{\mathsf{Nat}}(s, x\, y.t, n);\sigma\rangle \Downarrow \langle v;\sigma''\rangle}$$

$$\frac{\langle n;\sigma\rangle \Downarrow \langle \mathsf{suc}\, v;\sigma'\rangle \qquad \langle \mathsf{rec}_{\mathsf{Nat}}(s, x\, y.t, v);\sigma'\rangle \Downarrow \langle v';\sigma''\rangle \qquad \langle t[v/x, v'/y];\sigma''\rangle \Downarrow \langle w;\sigma'''\rangle}{\langle \mathsf{rec}_{\mathsf{Nat}}(s, x\, y.t, n);\sigma\rangle \Downarrow \langle w;\sigma'''\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}{\langle \mathsf{now}\, t;\sigma\rangle \Downarrow \langle \mathsf{now}\, v;\sigma'\rangle} \qquad \frac{\langle t_1;\sigma\rangle \Downarrow \langle v_1;\sigma'\rangle \qquad \langle t_2;\sigma'\rangle \Downarrow \langle v_2;\sigma''\rangle}{\langle \mathsf{wait}\, t_1\, t_2;\sigma\rangle \Downarrow \langle \mathsf{wait}\, v_1\, v_2;\sigma''\rangle}$$

$$\frac{\langle u;\sigma\rangle \Downarrow \langle \mathsf{now}\, v;\sigma'\rangle \qquad \langle s[v/x];\sigma'\rangle \Downarrow \langle w;\sigma''\rangle}{\langle \mathsf{rec}_{\mathcal{U}}(x.s, x\, y\, z.t, u);\sigma\rangle \Downarrow \langle w;\sigma''\rangle}$$

$$\frac{\begin{array}{c}\langle u;\sigma\rangle \Downarrow \langle \mathsf{wait}\, v_1\, v_2;\sigma'\rangle \\ \langle t[v_1/x, v_2/y, l/z];(\sigma', l\mapsto \mathsf{rec}_{\mathcal{U}}(x.s, x\, y\, z.t, \mathsf{adv}(v_2)))\rangle \Downarrow \langle v';\sigma''\rangle \qquad l = \mathsf{alloc}\,(\sigma')\end{array}}{\langle \mathsf{rec}_{\mathcal{U}}(x.s, x\, y\, z.t, u);\sigma\rangle \Downarrow \langle v';\sigma''\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}{\langle \mathsf{into}\, t;\sigma\rangle \Downarrow \langle \mathsf{into}\, v;\sigma'\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow \langle \mathsf{into}\, v;\sigma'\rangle}{\langle \mathsf{out}\, t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}$$

$$\frac{\langle t;\bullet\rangle \Downarrow \langle \mathsf{fix}\, x.t';\bullet\rangle \qquad \langle t'[\mathsf{box}(\mathsf{delay}(\mathsf{unbox}(\mathsf{fix}\, x.t')))/x];\sigma\rangle \Downarrow \langle v;\sigma'\rangle \qquad \sigma\neq\bullet}{\langle \mathsf{unbox}\, t;\sigma\rangle \Downarrow \langle v;\sigma'\rangle}$$

Fig. 4. Evaluation semantics.

$$\frac{\langle t;\eta\checkmark\rangle \Downarrow \langle v :: w;\eta_N\checkmark\eta_L\rangle}{\langle t;\eta\rangle \overset{v}{\Longrightarrow}_{\mathsf{Str}} \langle \mathsf{adv}\, w;\eta_L\rangle} \qquad \frac{\langle t;\eta\checkmark\rangle \Downarrow \langle \mathsf{wait}\, v\, w;\eta_N\checkmark\eta_L\rangle}{\langle t;\eta\rangle \overset{v}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{adv}\, w;\eta_L\rangle} \qquad \frac{\langle t;\eta\checkmark\rangle \Downarrow \langle \mathsf{now}\, v;\eta_N\checkmark\eta_L\rangle}{\langle t;\eta\rangle \overset{v}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT};\eta_L\rangle}$$

$$\frac{\langle t;\eta\rangle \overset{v}{\Longrightarrow}_{\mathcal{U}} \langle t';\eta'\rangle}{\langle t;\eta;p\rangle \overset{\mathsf{in}_p\, v}{\Longrightarrow}_{\mathsf{F}} \langle t';\eta';p\rangle} \qquad \frac{\langle t;\eta\rangle \overset{\langle v,w\rangle}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT};\eta'\rangle}{\langle t;\eta;1\rangle \overset{\mathsf{in}_2\, v}{\Longrightarrow}_{\mathsf{F}} \langle \mathsf{adv}\, w;\eta';2\rangle} \qquad \frac{\langle t;\eta\rangle \overset{\langle v,w\rangle}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT};\eta'\rangle}{\langle t;\eta;2\rangle \overset{\mathsf{in}_1\, v}{\Longrightarrow}_{\mathsf{F}} \langle \mathsf{out}(\mathsf{adv}\, w);\eta';1\rangle}$$

Fig. 5. Step semantics for streams, until types and fair streams.

The operational semantics of the guarded fixed point combinator fix closely follows the intuition provided by its type: The fixed point $\mathsf{fix}\, x.t$ is unfolded by delaying it into the future and substituting it for x in t. However, since $\mathsf{fix}\, x.t$ is of type $\Box A$, it first has to be unboxed before the delay and boxed again afterwards.

The recursion principle for $\mathcal{U}$ types is similar to the primitive recursion principle one would obtain for an inductive type $\mu\alpha.B + (A \times \alpha)$. The difference to an $\mathcal{U}$ type, i.e., a type $\mu\alpha.B + (A \times \bigcirc\alpha)$, is that each recursive call $\mathsf{rec}_{\mathcal{U}}(\dots)$ is shifted one time step into the future by placing it in the heap. As opposed to fix, however, no additional unboxing and re-boxing is required.

4.2 Step Semantics

The *step semantics* given in Figure 5 describes the computation performed by a Lively RaTT program *over time*. The notation $\langle t;\eta\rangle \overset{v}{\Longrightarrow}_{\mathsf{Str}} \langle t';\eta'\rangle$ indicates the passage of one time step during which a configuration consisting of a stream program t and a heap η transitions to the program t' and heap η' emitting the output v. We give three separate step semantics for stream, until, and fair stream types, denoted $\Longrightarrow_{\mathsf{Str}}$, $\Longrightarrow_{\mathcal{U}}$, and $\Longrightarrow_{\mathsf{F}}$, respectively. In addition, we state our metatheoretic results: $\Longrightarrow_{\mathsf{Str}}$ is productive, $\Longrightarrow_{\mathcal{U}}$ is guaranteed to terminate, and $\Longrightarrow_{\mathsf{F}}$ indeed produces a *fair* stream. But we defer the proofs of these results to section 5.

A closed term t of type $\Box\mathsf{Str}(A)$ is meant to produce an infinite stream $v_1, v_2, v_3, \dots$ of values of type A in a step-by-step fashion:

$$\langle \mathsf{unbox}\, t;\emptyset\rangle \overset{v_1}{\Longrightarrow}_{\mathsf{Str}} \langle t_1;\eta_1\rangle \overset{v_2}{\Longrightarrow}_{\mathsf{Str}} \langle t_2;\eta_2\rangle \overset{v_3}{\Longrightarrow}_{\mathsf{Str}} \cdots$$

Each step $\langle t_i;\eta_i\rangle \overset{v_i}{\Longrightarrow}_{\mathsf{Str}} \langle t_{i+1};\eta_{i+1}\rangle$ proceeds by first evaluating $\langle t_i;\eta_i\checkmark\rangle$ to $\left\langle v_i :: w;\eta_i'\checkmark\eta_{i+1}\right\rangle$, i.e., the head $v_i : A$ and the tail $w : \triangleright\mathsf{Str}(A)$ of the stream. Computation may then proceed in the next time step with the term $t_{i+1} = \mathsf{adv}\, w$ and heap η_{i+1} The old heap η_i', which consists of η_i possibly extended during evaluation of t_i, is garbage collected.

We can show that a term of type $\Box\mathsf{Str}(A)$ indeed produces such an infinite sequence of outputs. To state the productivity property of streams concisely, we restrict ourselves to streams over *value types*, which are described by the following grammar:

$$U, V ::= 1 \mid \mathsf{Nat} \mid U \times V \mid U + V$$

Theorem 4.1 (Productivity)**.** If $\vdash t : \Box\mathsf{Str}(A)$, then there is an infinite sequence of reduction steps

$$\langle \mathsf{unbox}\, t;\emptyset\rangle \overset{v_1}{\Longrightarrow}_{\mathsf{Str}} \langle t_1;\eta_1\rangle \overset{v_2}{\Longrightarrow}_{\mathsf{Str}} \langle t_2;\eta_2\rangle \overset{v_3}{\Longrightarrow}_{\mathsf{Str}} \cdots$$

Moreover, if A is a value type, then $\vdash v_i : A$ for all $i \geq 1$.

In particular, this means that the machine will never get stuck trying to access a heap location that has been garbage collected. In other words, the aggressive garbage collection of the heap used in the step semantics is safe.

Intuitively speaking, value types describe static, time independent data, and therefore exclude functions and modal types. Since A is a value type in the above theorem, we can give a concise characterisation of the output produced by the stream in terms of the syntactic typing $\vdash v_i : A$.

The operational semantics for until types proceeds similarly to stream types, but has an additional case for when the computation eventually halts by evaluating to a value of the form $\mathsf{now}\, v$.

We can show that a term of type $\Box(A \;\mathcal{U}\; B)$ produces a sequence of values of type A, but eventually halts with a value of type B:

Theorem 4.2 (Termination)**.** If $\vdash t : \Box(A \;\mathcal{U}\; B)$, then there is a finite sequence of reduction steps

$$\langle \mathsf{unbox}\, t; \emptyset\rangle \overset{v_1}{\Longrightarrow}_{\mathcal{U}} \langle t_1; \eta_1\rangle \overset{v_2}{\Longrightarrow}_{\mathcal{U}} \langle t_2; \eta_2\rangle \overset{v_2}{\Longrightarrow}_{\mathcal{U}} \ldots \overset{v_n}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT}; \eta_n\rangle$$

Moreover, if A and B are value types, then $\vdash v_i : A$ for all $0 < i < n$, and $\vdash v_n : B$.

The computation performed by a fair stream of type $\Box\mathsf{Fair}(\mathsf{A}, \mathsf{B})$ requires a bit of additional bookkeeping: The state of the machine is represented by a triple $\langle t; \eta; p\rangle$ consisting of a term t, a heap η and a value $p \in \{1, 2\}$ that indicates the mode that the computation represented by t is in. If $p = 1$, then the most recent output was of type A, whereas $p = 2$ indicates that the most recent output was of type B. A fair execution thus means that the computation may not remain in the same mode indefinitely.

Theorem 4.3 (Liveness)**.** If $\vdash t : \Box\mathsf{Fair}(\mathsf{A}, \mathsf{B})$, then there is an infinite sequence of reduction steps

$$\langle \mathsf{out}\,(\mathsf{unbox}\, t); \emptyset; 1\rangle \overset{\mathsf{in}_{p_1} v_1}{\Longrightarrow}_{\mathsf{F}} \langle t_1; \eta_1; p_1\rangle \overset{\mathsf{in}_{p_2} v_2}{\Longrightarrow}_{\mathsf{F}} \langle t_2; \eta_2; p_2\rangle \overset{\mathsf{in}_{p_3} v_3}{\Longrightarrow}_{\mathsf{F}} \ldots$$

such that for each $p \in \{1, 2\}$, we have that $p_i = p$ for infinitely many $i \geq 1$. Moreover, if A and B are value types, then $\vdash \mathsf{in}_{p_i} v_i : A + B$ for all $i \geq 1$.

As a special case of the fair stream type we obtain the type $\mathsf{Live}(A) = \mathsf{Fair}(1, \mathsf{A})$. Terms of this type do not produce a result at every time step but they will produce infinitely many results.

4.3 Reactive Step Semantics

The step semantics in section 4.2 captures closed computations without input from an external environment. To capture reactive computations, we give for each step semantics $\Longrightarrow_M$, a corresponding *reactive* step semantics in Figure 6. For instance, the reactive step semantics for streams may at each step consume some input v and produce an output v', which we denote by $\overset{v/v'}{\Longrightarrow}_{\mathsf{Str}}$.

To supply input to the computation, the machine configurations for the reactive step semantics are extended by an additional component l, which is the heap location from where the next input value can be retrieved. For instance, the reactive step semantics for streams takes a step $\langle t; \eta; l\rangle \overset{v/v'}{\Longrightarrow}_{\mathsf{Str}} \langle t'; \eta'; l'\rangle$ by placing the value $v :: l'$ in the heap location location l, where l' is a freshly allocated heap location that serves as a stand-in for the subsequent input. Assigning l' the dummy value $\langle\rangle$ will prevent the machine from allocating l' during evaluation of t. The reactive step semantics for $\mathcal{U}$-types and fair streams follow the same pattern.

Given a term t of type $\Box(\mathsf{Str}(A) \to \mathsf{Str}(B))$, and a sequence $v_1, v_2, \ldots$ of values of type A, there is an infinite sequence of reduction steps

$$\langle \mathsf{unbox}\, t\,(\mathsf{adv}\, l_0); \emptyset; l_0\rangle \overset{v_1/v_1'}{\Longrightarrow}_{\mathsf{Str}} \langle t_1; \eta_1; l_1\rangle \overset{v_2/v_2'}{\Longrightarrow}_{\mathsf{Str}} \langle t_2; \eta_2; l_2\rangle \overset{v_3/v_3'}{\Longrightarrow}_{\mathsf{Str}} \cdots$$

$$\frac{\langle t;\eta, l \mapsto v :: l' \checkmark l' \mapsto \langle\rangle\rangle \Downarrow \langle v' :: w; \eta_N \checkmark \eta_L, l' \mapsto \langle\rangle\rangle \qquad l' = \mathsf{alloc}\,(\eta\checkmark)}{\langle t;\eta;l\rangle \overset{v/v'}{\Longrightarrow}_{\mathsf{Str}} \langle \mathsf{adv}\, w; \eta_L; l'\rangle}$$

$$\frac{\langle t;\eta, l \mapsto v :: l' \checkmark l' \mapsto \langle\rangle\rangle \Downarrow \langle \mathsf{wait}\, v'\, w; \eta_N \checkmark \eta_L, l \mapsto \langle\rangle\rangle \qquad l' = \mathsf{alloc}\,(\eta\checkmark)}{\langle t;\eta;l\rangle \overset{v/v'}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{adv}\, w; \eta_L; l'\rangle}$$

$$\frac{\langle t;\eta, l \mapsto v :: l' \checkmark l' \mapsto \langle\rangle\rangle \Downarrow \langle \mathsf{now}\, v'; \eta_N \checkmark \eta_L, l' \mapsto \langle\rangle\rangle \qquad l' = \mathsf{alloc}\,(\eta\checkmark)}{\langle t;\eta;l\rangle \overset{v/v'}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT}; \eta_L; l'\rangle}$$

$$\frac{\langle t;\eta;l\rangle \overset{v/v'}{\Longrightarrow}_{\mathcal{U}} \langle t';\eta';l'\rangle}{\langle t;\eta;l;p\rangle \overset{v/\mathsf{in}_p\, v'}{\Longrightarrow}_{\mathsf{F}} \langle t';\eta';l';p\rangle} \qquad \frac{\langle t;\eta;l\rangle \overset{v/\langle v',w\rangle}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT};\eta';l'\rangle}{\langle t;\eta;l;1\rangle \overset{v/\mathsf{in}_2\, v'}{\Longrightarrow}_{\mathsf{F}} \langle \mathsf{adv}\, w;\eta';l';2\rangle}$$

$$\frac{\langle t;\eta;l\rangle \overset{v/\langle v',w\rangle}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT};\eta';l'\rangle}{\langle t;\eta;l;2\rangle \overset{v/\mathsf{in}_1\, v'}{\Longrightarrow}_{\mathsf{F}} \langle \mathsf{out}(\mathsf{adv}\, w);\eta';l';1\rangle}$$

Fig. 6. Reactive step semantics for streams, until types and fair streams.

such that $\vdash v_i : B$ for all $i \geq 1$. The first term of the computation sets up the initial promise of an input by giving the term $\mathsf{unbox}\, t$ of type $\mathsf{Str}(A) \to \mathsf{Str}(B)$ the argument $\mathsf{adv}\, l_0$. The location l_0 is simply the first fresh heap location, i.e. $l_0 = \mathsf{alloc}\,(\emptyset)$; While $\mathsf{adv}\, l_0$ is not a well-typed term, it has the type $\mathsf{Str}(A)$ *semantically*, in the sense that $\mathsf{adv}\, l_0$ is a term in the logical relation $[\![\mathsf{Str}(A)]\!]$ that we construct in section 5.

Theorem 4.4 (Causality)**.** Let $v_1, v_2, \ldots$ be an infinite sequence of values with $\vdash v_i : A$ for all $i \geq 1$.

(i) If $\vdash t : \Box(\mathsf{Str}(A) \to \mathsf{Str}(B))$, then there is an infinite sequence of reduction steps

$$\langle \mathsf{unbox}\, t\, (\mathsf{adv}\, l_0); \emptyset; l_0\rangle \overset{v_1/v_1'}{\Longrightarrow}_{\mathsf{Str}} \langle t_1;\eta_1;l_1\rangle \overset{v_2/v_2'}{\Longrightarrow}_{\mathsf{Str}} \langle t_2;\eta_2;l_2\rangle \overset{v_3/v_3'}{\Longrightarrow}_{\mathsf{Str}} \cdots$$

Moreover, if B is a value type, then $\vdash v_i' : B$ for all $i \geq 1$.

(ii) If $\vdash t : \Box(\mathsf{Str}(A) \to B\ \mathcal{U}\ C)$, then there is a finite sequence of reduction steps

$$\langle \mathsf{unbox}\, t\, (\mathsf{adv}\, l_0); \emptyset; l_0\rangle \overset{v_1/v_1'}{\Longrightarrow}_{\mathcal{U}} \langle t_1;\eta_1;l_1\rangle \overset{v_2/v_2'}{\Longrightarrow}_{\mathcal{U}} \langle t_2;\eta_2;l_2\rangle \overset{v_3/v_3'}{\Longrightarrow}_{\mathcal{U}} \ldots \overset{v_n/v_n'}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT};\eta_n;l_n\rangle$$

Moreover, if B and C are value types, then $\vdash v_i' : B$ for all $0 < i < n$, and $\vdash v_n' : C$.

(iii) If $\vdash t : \Box(\mathsf{Str}(A) \to \mathsf{Fair}(\mathsf{B},\mathsf{C}))$, then there is an infinite sequence of reduction steps

$$\langle \mathsf{out}\, (\mathsf{unbox}\, t\, (\mathsf{adv}\, l_0)); \emptyset; l_0; 1\rangle \overset{v_1/\mathsf{in}_{p_1}\, v_1'}{\Longrightarrow}_{\mathsf{F}} \langle t_1;\eta_1;l_1;p_1\rangle \overset{v_2/\mathsf{in}_{p_2}\, v_2'}{\Longrightarrow}_{\mathsf{F}} \langle t_2;\eta_2;l_2;p_2\rangle \overset{v_3/\mathsf{in}_{p_3}\, v_3'}{\Longrightarrow}_{\mathsf{F}} \ldots$$

such that for each $p \in \{1,2\}$, we have that $p_i = p$ for infinitely many $i \geq 1$. Moreover, if B and C are value types, then $\vdash \mathsf{in}_{p_i} v_i' : B + C$ for all $i \geq 1$.

Since the operational semantics is deterministic, in each step $\langle t_i;\eta_i;l_i\rangle \overset{v_{i+1}/v_{i+1}'}{\Longrightarrow}_{\mathsf{Str}} \langle t_{i+1};\eta_{i+1};l_{i+1}\rangle$ the resulting output v_{i+1}' and new state of the computation $\langle t_{i+1};\eta_{i+1};l_{i+1}\rangle$ are uniquely determined by the previous state $\langle t_i;\eta_i;l_i\rangle$ and the input v_{i+1}. Thus, v_{i+1}' and $\langle t_{i+1};\eta_{i+1};l_{i+1}\rangle$ are independent

of future inputs v_j with $j > i+1$. The same is true for the corresponding reactive step semantics of $\mathcal{U}$-types and fair streams.

5 METATHEORY

In this section we show the soundness of the type system, which typically means that a well-typed term will never get stuck. However, we show a stronger, *semantic* type soundness property that will allow us to prove the operational properties detailed in section 4. To this end, we devise a Kripke logical relation. Essentially, such a logical relation is a family $[\![A]\!](w)$ of sets of closed terms that satisfy the desired soundness property. This family of sets is indexed by w drawn from a suitable sets of "worlds" and defined inductively on the structure of the type A and world w. The proof of soundness is then reduced to a proof that $\vdash t : A$ implies $t \in [\![A]\!](w)$.

5.1 Worlds

To a first approximation, the worlds in our logical relation contain two ordinals $\alpha \leqslant \omega$ and $\beta < \omega \cdot 2$ and a store σ. The two ordinals are used to define the logical relation for recursive types, the first for temporal inductive types and the latter for step-indexed guarded recursive types. For guarded recursive types, we achieve this by defining $[\![\triangleright A]\!](\sigma, \alpha, \beta)$ in terms of $[\![A]\!](\sigma, \alpha, \beta')$ for strictly smaller β'. Since unfolding $\mathsf{Fix}\,\alpha.A$ introduces a $\triangleright$, the step index decreases for the recursive call. For the inductive types, we define $[\![A \,\mathcal{U}\, B]\!](\sigma, \alpha, \beta)$ in terms of $[\![\bigcirc(A \,\mathcal{U}\, B)]\!](\sigma, \alpha', \beta)$ where $\alpha' < \alpha$. Intuitively, α gives an upper limit to the number of unfoldings of the inductive type used in terms.

While this setup is sufficient for proving productivity, safety of garbage collection, termination, and liveness properties, it is not enough to capture causality. To characterise causality, the logical relation also needs to account for the possible inputs a given term may receive. We do this by further indexing the logical relation by a sequence of future inputs. To this end, we assume an infinite sequence of heaps $\eta_1; \eta_2; \ldots$, denoted $\overline{\eta}$, that describes the input that is received at each point in the future. The namespaces of all heaps in $\overline{\eta}$ and σ are assumed pairwise disjoint. The worlds in our logical relation are thus of the from $(\sigma, \overline{\eta}, \alpha, \beta)$.

As is standard for Kripke logical relations, our relation will be closed under moving to a bigger world. Worlds are ordered as follows: $(\sigma, \overline{\eta}, \alpha, \beta) \leqslant (\sigma', \overline{\eta}', \alpha', \beta')$ if $\sigma \sqsubseteq_{\checkmark} \sigma'$, $\overline{\eta} \sqsubseteq \overline{\eta}'$, $\alpha = \alpha'$ and $\beta' \leqslant \beta$. Here $\overline{\eta} \sqsubseteq \overline{\eta}'$ refers to the pointwise ordering on partial maps (assuming identity of namespaces) and $\sigma \sqsubseteq_{\checkmark} \sigma'$ is the extension of this with the rule $\eta_L \sqsubseteq_{\checkmark} \eta_N \checkmark \eta_L$ to a preorder. Note that the null store $\bullet$ is only related to itself under this ordering.

5.2 Support and Renamings

To prove closure of the Kripke semantics under store extensions, a similar property must be proved for the machine, i.e, if $\langle t; \sigma \rangle$ evaluates to a value and if $\sigma \sqsubseteq_{\checkmark} \sigma'$ then $\langle t; \sigma' \rangle$ evaluates to the same value. However, this statement is not entirely true, because the machine, when run in an extended state, may allocate different locations on the heap and the resulting values may differ correspondingly. To prove that this is the only way that the values can differ, we introduce notions of a renaming and support. Similar notions were used in the model by Krishnaswami [2013].

A renaming is a map $\phi : \mathsf{Loc} \to \mathsf{Loc}$ respecting name spaces. Such a map acts on terms by substitution, and this extends to heaps and stores by $\phi(\eta, l \mapsto t) = \phi(\eta), \phi(l) \mapsto \phi(t)$. We write $\phi : (t, \sigma, \overline{\eta}) \to (t', \sigma', \overline{\eta}')$ if $\phi(t) = t', \phi(\sigma) \sqsubseteq_{\checkmark} \sigma', \phi(\overline{\eta}) \sqsubseteq \overline{\eta}'$. Given a term t and a pair of a store and a heap sequence $(\sigma, \overline{\eta})$, we say that t is *supported* by $(\sigma, \overline{\eta})$, written $t \bowtie (\sigma, \overline{\eta})$, if whenever a location in t occurs in the namespaces of σ or $\overline{\eta}$, it must be in the domain of σ or $\overline{\eta}$, respectively. Given $(\sigma, \overline{\eta})$, we write $(\sigma, \overline{\eta})$ $\mathsf{supported}$ if all values in the codomains of σ and $\overline{\eta}$ are supported by $(\sigma, \overline{\eta})$. We restrict attention to Kripke worlds where $(\sigma, \overline{\eta})$ is supported.

5.3 Logical Relation

Our logical relation consists of two parts: A *value relation* $\mathcal{V}[\![A]\!](w)$ that contains all values that semantically inhabit type A at the world w, and a corresponding *term relation* $\mathcal{T}[\![A]\!](w)$ containing terms that evaluate to elements in $\mathcal{V}[\![A]\!](w)$. The two relations are defined by mutual induction in Figure 7. More precisely, the two relations are defined by well-founded recursion by the lexicographic ordering on the tuple $(\beta, |A|, \alpha, e)$, where $|A|$ is the size of A defined below, and $e = 1$ for the term relation and $e = 0$ for the value relation.

$$\begin{aligned} |\alpha| = |\triangleright A| = |1| = |\mathsf{Nat}| &= 1 \\ |A \times B| = |A + B| = |A \mathbin{\mathcal{U}} B| = |A \to B| &= 1 + |A| + |B| \\ |\Box A| = |\bigcirc A| = |\mathsf{Fix}\,\alpha.A| &= 1 + |A| \end{aligned}$$

Note that since $|\triangleright(\mathsf{Fix}\ \alpha.A)| = |\alpha|$, the size of $A[\triangleright\mathsf{Fix}\,\alpha.A/\alpha]$ is strictly smaller than that of $\mathsf{Fix}\,\alpha.A$, which justifies the well-foundedness of recursive types. Note also that $\mathcal{V}[\![A \mathbin{\mathcal{U}} B]\!](\sigma, \overline{\eta}, \alpha, \beta)$ is defined in terms of $\mathcal{V}[\![\bigcirc(A \mathbin{\mathcal{U}} B)]\!](\sigma, \overline{\eta}, \alpha', \beta)$ for $\alpha' < \alpha$. To obtain well-foundedness, we would need $|\bigcirc(A \mathbin{\mathcal{U}} B)| \leqslant |A \mathbin{\mathcal{U}} B|$, which is not true. But this problem can be avoided by "inlining" the definition of $\mathcal{V}[\![\bigcirc(A \mathbin{\mathcal{U}} B)]\!](\sigma, \overline{\eta}, \alpha', \beta)$, which is defined in terms of $\mathcal{T}[\![A \mathbin{\mathcal{U}} B]\!](\sigma', \overline{\eta}', \alpha', \beta)$ where σ' and $\overline{\eta}'$ are appropriately modified. For the sake of readability, we will keep the definitions as given.

In the definition for $A \to B$ we explicitly add the closure properties for support, renaming and worlds. Further, we restrict the lambda abstractions to garbage collected stores. This reflects the typing rule for lambda abstractions, which requires Γ to be tick-free.

The definition of $\Box A$ captures the notion of stability. All terms must be free of locations and able to evaluate safely with any input sequence and hence, in any future.

The value relations for $\triangleright A$ and $\bigcirc A$ differ only in the case where β is a limit ordinal. In the successor case, they encapsulate the soundness of garbage collection: The set $\mathcal{V}[\![m\ a]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta+1), m \in \{\bigcirc, \triangleright\}$ contains all heap locations that can be read and executed safely in the next timestep. In particular, such terms must evaluate in the garbage collected store extended with the next set of external inputs. If β is a limit ordinal, $\bigcirc A$ has the same interpretation except that the index β is fixed. This is to ensure that inductive types have the correct global behaviour. On the other hand, $\triangleright A$ is defined to be the intersection of the interpretation at all smaller (β)-indices. This definition is needed for the interpretation of fixed points.

In the definition of $A \mathbin{\mathcal{U}} B$ we see the use of the α-index to give an upper bound on the number of unfoldings used in the elements of the logical relation. In particular, if $\alpha = 0$, the relation contains only values of the form $\mathsf{now}\, v$ whereas if $\alpha > 0$, the relation also contain values of the form $\mathsf{wait}\, u\, w$ defined in terms of values from $\mathcal{V}[\![\bigcirc(A \mathbin{\mathcal{U}} B)]\!](\sigma, \overline{\eta}, \alpha', \beta)$ where $\alpha' < \alpha$.

Our value and term interpretation is closed w.r.t the Kripke structure on $\sigma, \overline{\eta}$ and β and the value relation is upwards closed w.r.t α for $\mathcal{U}$-types.

Lemma 5.1 (Kripke Properties). *Given A, B and worlds $(\sigma, \overline{\eta}, \alpha, \beta), (\sigma', \overline{\eta}', \alpha', \beta')$ s.t $\sigma \sqsubseteq_{\checkmark} \sigma', \overline{\eta} \sqsubseteq \overline{\eta}', \alpha \leqslant \alpha'$ and $\beta' \leqslant \beta$ we have*

(1) $\mathcal{V}[\![A]\!](\sigma, \overline{\eta}, \alpha, \beta) \subseteq \mathcal{V}[\![A]\!](\sigma', \overline{\eta}', \alpha, \beta')$
(2) $\mathcal{T}[\![A]\!](\sigma, \overline{\eta}, \alpha, \beta) \subseteq \mathcal{T}[\![A]\!](\sigma', \overline{\eta}', \alpha, \beta')$
(3) $\mathcal{V}[\![A \mathbin{\mathcal{U}} B]\!](\sigma, \overline{\eta}, \alpha, \beta) \subseteq \mathcal{V}[\![A \mathbin{\mathcal{U}} B]\!](\sigma, \overline{\eta}, \alpha', \beta)$

As stated above we treat $\bigcirc$ as a sub-modality of $\triangleright$ and this is expressed semantically in the following lemma:

$$
\begin{aligned}
\mathcal{V}[\![1]\!](w) &= \{\langle\rangle\}, \\
\mathcal{V}[\![\mathsf{Nat}]\!](w) &= \{\mathsf{suc}^n\, 0 \mid n \in \mathbb{N}\}, \\
\mathcal{V}[\![A \times B]\!](w) &= \left\{(v_1, v_2) \,\middle|\, v_1 \in \mathcal{V}[\![A]\!](w) \wedge v_2 \in \mathcal{V}[\![B]\!](w)\right\}, \\
\mathcal{V}[\![A + B]\!](w) &= \left\{\mathsf{in}_1\, v \,\middle|\, v \in \mathcal{V}[\![A]\!](w)\right\} \cup \left\{\mathsf{in}_2\, v \,\middle|\, v \in \mathcal{V}[\![B]\!](w)\right\} \\
\mathcal{V}[\![A \to B]\!](\sigma, \overline{\eta}, \alpha, \beta) &= \left\{\lambda x.t \,\middle|\, t \bowtie (\sigma, \overline{\eta}) \wedge \forall \beta' \le \beta. \forall \psi : (t, \sigma, \overline{\eta}) \to (t', \sigma', \overline{\eta}').\right. \\
&\qquad \left. \forall v \in \mathcal{V}[\![A]\!](\mathsf{gc}\,(\sigma'), \overline{\eta}', \alpha, \beta').t'[v/x] \in \mathcal{T}[\![B]\!](\mathsf{gc}\,(\sigma'), \overline{\eta}', \alpha, \beta')\right\} \\
\mathcal{V}[\![\Box A]\!](\sigma, \overline{\eta}, \alpha, \beta) &= \left\{t \,\middle|\, \forall \overline{\eta}'.\mathsf{unbox}\; t \in \mathcal{T}[\![A]\!](\emptyset, \overline{\eta}', \alpha, \beta) \wedge t \;\mathsf{location\text{-}free}\right\} \\
\mathcal{V}[\![\bigcirc A]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta) &= \begin{cases} \mathsf{dom}\,(\mathsf{gc}\,(\sigma)) & \beta = 0 \wedge \sigma \neq \bullet \\ \left\{l \,\middle|\, \mathsf{adv}\, l \in \mathcal{T}[\![A]\!](\mathsf{gc}\,(\sigma) \checkmark \eta, \overline{\eta}, \alpha, \beta')\right\} & \beta = \beta' + 1 \wedge \sigma \neq \bullet \\ \left\{l \,\middle|\, \mathsf{adv}\, l \in \mathcal{T}[\![A]\!](\mathsf{gc}\,(\sigma) \checkmark \eta, \overline{\eta}, \alpha, \beta)\right\} & \beta \text{ limit ordinal} \wedge \sigma \neq \bullet \end{cases} \\
\mathcal{V}[\![\triangleright A]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta) &= \begin{cases} \mathsf{dom}\,(\mathsf{gc}\,(\sigma)) & \beta = 0 \wedge \sigma \neq \bullet \\ \left\{l \,\middle|\, \mathsf{adv}\, l \in \mathcal{T}[\![A]\!](\mathsf{gc}\,(\sigma) \checkmark \eta, \overline{\eta}, \alpha, \beta')\right\} & \beta = \beta' + 1 \wedge \sigma \neq \bullet \\ \bigcap_{\beta' < \beta} \mathcal{V}[\![\triangleright A]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta') & \beta \text{ limit ordinal} \wedge \sigma \neq \bullet \end{cases} \\
\mathcal{V}[\![\mathsf{Fix}\,\alpha.A]\!](w) &= \left\{\mathsf{into}(v) \,\middle|\, v \in \mathcal{V}[\![A[\triangleright(\mathsf{Fix}\,\alpha.A)/\alpha]]\!](w)\right\} \\
\mathcal{V}[\![A\ \mathcal{U}\ B]\!](\sigma, \overline{\eta}, \alpha, \beta) &= \left\{\mathsf{now}\, v \,\middle|\, v \in \mathcal{V}[\![B]\!](\sigma, \overline{\eta}, \omega, \beta)\right\} \cup \\
&\quad \left\{\mathsf{wait}\, v\, w \,\middle|\, v \in \mathcal{V}[\![A]\!](\sigma, \overline{\eta}, \omega, \beta)\right. \\
&\qquad \left. \wedge\ \exists \alpha' < \alpha. w \in \mathcal{V}[\![\bigcirc(A\ \mathcal{U}\ B)]\!](\sigma, \overline{\eta}, \alpha', \beta)\right\} \\
\mathcal{T}[\![A]\!](\sigma, \overline{\eta}, \alpha, \beta) &= \left\{t \,\middle|\, t \bowtie (\sigma, \overline{\eta}) \wedge \exists \sigma', v.\, \langle t; \sigma\rangle \Downarrow \langle v; \sigma'\rangle \wedge v \in \mathcal{V}[\![A]\!](\sigma', \overline{\eta}, \alpha, \beta)\right\}
\end{aligned}
$$

$$
\textsc{Garbage collection:} \qquad \mathsf{gc}\,(\bullet) = \bullet \qquad \mathsf{gc}\,(\eta_L) = \eta_L \qquad \mathsf{gc}\,(\eta_N \checkmark \eta_L) = \eta_L
$$

Fig. 7. Logical Relation.

Lemma 5.2 (Sub-modality). *Given A and world $(\sigma, \overline{\eta}, \alpha, \beta)$ then*

$$\mathcal{V}[\![\bigcirc A]\!](\sigma, \overline{\eta}, \alpha, \beta) \subseteq \mathcal{V}[\![\triangleright A]\!](\sigma, \overline{\eta}, \alpha, \beta)$$

Proof. Follows by transfinite induction on β. □

The next lemma justifies the terminology 'limit types', by showing that the interpretation of these at limit ordinals is the intersection of the interpretations at the ordinals below. In category theoretic terms, the intersection is a limit, and such a type is a sheaf [MacLane and Moerdijk 2012].

Lemma 5.3 (Limit Types). *If A limit and β is a limit ordinal, then*

$$\bigcap_{\beta' < \beta} \mathcal{V}[\![A]\!](\sigma, \overline{\eta}, \alpha, \beta') = \mathcal{V}[\![A]\!](\sigma, \overline{\eta}, \alpha, \beta) \qquad \bigcap_{\beta' < \beta} \mathcal{T}[\![A]\!](\sigma, \overline{\eta}, \alpha, \beta') = \mathcal{T}[\![A]\!](\sigma, \overline{\eta}, \alpha, \beta)$$

Proof. In the first equality, the inclusion from right to left follows from Lemma 5.1, and the other inclusion is proved by induction on A. The second equality then follows from the first. □

In the special case where A is a limit type, $\bigcirc$ and $\triangleright$ do in fact coincide:

Corollary 5.4 (Sub-modality at limit). *Given A and a world w s.t. A limit, then*

$$\mathcal{V}[\![\bigcirc A]\!](w) = \mathcal{V}[\![\triangleright A]\!](w)$$

$$
\begin{aligned}
C[\![\cdot]\!](\bullet, \overline{\eta}, \beta) &= \{*\} \\
C[\![\Gamma, x : A]\!](\sigma, \overline{\eta}, \beta) &= \left\{ \gamma[x \mapsto v] \,\middle|\, \gamma \in C[\![\Gamma]\!](\sigma, \overline{\eta}, \beta), v \in \mathcal{V}[\![A]\!](\sigma, \overline{\eta}, \omega, \beta) \right\} \\
C[\![\Gamma, \checkmark_{\triangleright}]\!]((\eta_N \checkmark \eta_L), \overline{\eta}, \beta) &= C[\![\Gamma]\!](\eta_N, (\eta_L; \overline{\eta}), \beta + 1) \\
C[\![\Gamma, \checkmark_{\bigcirc}]\!]((\eta_N \checkmark \eta_L), \overline{\eta}, \beta) &= \begin{cases} C[\![\Gamma]\!](\eta_N, (\eta_L; \overline{\eta}), \beta) & \beta \text{ limit ordinal} \\ C[\![\Gamma]\!](\eta_N, (\eta_L; \overline{\eta}), \beta + 1) & \text{otherwise} \end{cases} \\
C[\![\Gamma, \sharp]\!](\sigma, \overline{\eta}, \beta) &= \bigcup_{\overline{\eta}'} C[\![\Gamma]\!](\bullet, \overline{\eta}', \beta) \qquad \sigma \neq \bullet
\end{aligned}
$$

Fig. 8. Context Relation

Proof. One inclusion always holds by Lemma 5.2, and the two sets are equal by definition except when β is a limit ordinal. In that case, by Lemma 5.3 it suffices to show that if $v \in \mathcal{V}[\![\triangleright A]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta)$ then $\mathsf{adv}\ v \in \mathcal{T}[\![A]\!]((\mathsf{gc}\,(\sigma) \checkmark \eta); \overline{\eta}, \alpha, \beta')$ for all $\beta' < \beta$, which follows from $v \in \mathcal{V}[\![\triangleright A]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta' + 1)$ □

Finally, we obtain the semantic soundness of the language phrased as the following fundamental property of the logical relation $\mathcal{T}[\![A]\!](\sigma, \overline{\eta}, \omega, \beta)$.

Theorem 5.5 (Fundamental Property)**.** If $\Gamma \vdash t : A$ and $\gamma \in C[\![\Gamma]\!](\sigma, \overline{\eta}, \beta)$, then $t\gamma \in \mathcal{T}[\![A]\!](\sigma, \overline{\eta}, \omega, \beta)$.

Here $C[\![\Gamma]\!](\sigma, \overline{\eta}, \beta)$ refers to the logical relation for typing contexts defined in Figure 8. Note the cases for $\Gamma, \checkmark_m$, which captures the intuition that variables occurring before $\checkmark_m$ arrive one time step before those to the right. Again $\bigcirc$ and $\triangleright$ differ only when β is a limit ordinal. Cases not mentioned in the figure (such as $C[\![\cdot]\!](\sigma, \overline{\eta}, \beta)$ for $\sigma \neq \bullet$) are interpreted as the empty set. The theorem is proved by a lengthy but entirely standard induction on the typing relation $\Gamma \vdash t : A$.

As an easy consequence of the fundamental property and the fact the empty substitution is an element of $C[\![\cdot, \sharp]\!](\sigma, \overline{\eta}, \beta)$ for any store σ and input sequence $\overline{\eta}$, we have the following property that we shall use to prove Lively RaTT's operational properties:

Corollary 5.6 (Fundamental Property)**.** *If $\sharp \vdash t : A$, then $t \in \mathcal{T}[\![A]\!](\sigma, \overline{\eta}, \omega, \beta)$ for all $\sigma, \overline{\eta}$ and β.*

5.4 Productivity, Termination, Liveness & Causality

In this section we demonstrate how we apply the fundamental property of the logical relation to prove the operational properties of Lively RaTT that we presented in section 4.2 and section 4.3. We have formulated these operational properties in terms of value types, so that we can use the following correspondence between semantic and syntactic typing:

Lemma 5.7. *Given any world w, value type A, and value v, we have that $v \in \mathcal{V}[\![A]\!](w)$ iff $\vdash v : A$.*

5.4.1 Productivity. We start with the productivity property of streams of type $\mathsf{Str}(A)$.

Given a type A and ordinal β, we define the following set of machine configurations for $\Longrightarrow_{\mathsf{Str}}$ that are safe according to the logical relation:

$$S(A, \beta) = \left\{ \langle t; \eta \rangle \,\middle|\, t \in \mathcal{T}[\![\mathsf{Str}(A)]\!](\eta \checkmark, \overline{\emptyset}, \omega, \beta) \right\}$$

where we use the notation $\overline{\emptyset}$ for the sequence of empty heaps with the appropriate namespace.

Intuitively speaking, a machine configuration c in $S(A, \beta)$ will be safe to execute for the next β steps of $\Longrightarrow_{\mathsf{Str}}$ and produces output of type A. We formulate the essence of the productivity property of such a stream as follows:

Lemma 5.8 (Productivity). *If* $\langle t; \eta\rangle \in S(A, \beta+1)$, *then there are* $\langle t'; \eta'\rangle \in S(A,\beta)$ *and* $v \in \mathcal{V}[\![A]\!](\eta', \overline{\emptyset}, \omega, \beta+1)$ *such that* $\langle t;\eta\rangle \overset{v}{\Longrightarrow}_{\mathsf{Str}} \langle t';\eta'\rangle$.

In each step of a stream computation, we count down by one on the step index β and produce an output v of semantic type A:

Proof of Theorem 4.1. By Corollary 5.6 we have that $\langle \mathsf{unbox}\, t; \emptyset\rangle \in S(A,\beta)$ for any β. Using Lemma 5.8, we can thus extend any finite reduction sequence

$$\langle \mathsf{unbox}\, t;\emptyset\rangle \overset{v_1}{\Longrightarrow}_{\mathsf{Str}} c_1 \overset{v_2}{\Longrightarrow}_{\mathsf{Str}} c_2 \overset{v_3}{\Longrightarrow}_{\mathsf{Str}} \cdots \overset{v_n}{\Longrightarrow}_{\mathsf{Str}} c_n$$

by an additional reduction step $c_n \overset{v_{n+1}}{\Longrightarrow}_{\mathsf{Str}} c_{n+1}$. Since $\Longrightarrow_{\mathsf{Str}}$ is deterministic, this uniquely defines the desired infinite reduction. Moreover, given that A is a value type, $\vdash v_i : A$ follows for all $i \geq 1$ by Lemma 5.7. □

5.4.2 Termination. Analogously to the set of machine states $S(A,\beta)$ for stream types, we define the following set $U(A,B,\alpha,\beta)$ for until types:

$$U(A,B,\alpha,\beta) = \left\{ \langle t;\eta\rangle \;\middle|\; t \in \mathcal{T}[\![A\ \mathcal{U}\ B]\!](\eta\checkmark, \overline{\emptyset}, \alpha, \beta) \right\}$$

This definition allows us to state the essence of the termination property for until types as follows:

Lemma 5.9 (Termination). *Given* $\langle t;\eta\rangle \in U(A,B,\alpha,\beta)$, *one of the following two statements holds:*

(a) There are t', η', $\alpha' < \alpha$, *and* $v \in \mathcal{V}[\![A]\!](\eta', \overline{\emptyset}, \omega, \beta)$ *such that*

$$\langle t;\eta\rangle \overset{v}{\Longrightarrow}_{\mathcal{U}} \langle t';\eta'\rangle \text{ and if } \beta > 0 \text{ then } \langle t';\eta'\rangle \in U(A,B,\alpha',\beta-1)$$

where $\beta - 1 = \beta'$ *if* $\beta = \beta' + 1$ *and otherwise* $\beta - 1 = \beta$.

(b) There is some $v \in \mathcal{V}[\![B]\!](\eta', \overline{\emptyset}, \omega, \beta)$ *such that* $\langle t;\eta\rangle \overset{v}{\Longrightarrow}_{\mathcal{U}} \langle \mathsf{HALT};\eta'\rangle$.

Theorem 4.2 is now an easy consequence of the above lemma and the fundamental property of the logical relation.

Proof of Theorem 4.2. By Corollary 5.6 $\mathsf{unbox}\, t \in U(A,B,\omega,\omega)$, and by Lemma 5.9 we can construct the desired sequence of reductions. Since the index α strictly decreases each time we take a step of the form (a), the sequence must eventually terminate with a step of the form (b). Moreover, by Lemma 5.7, the output values v_i have the desired type given that A and B are value types. □

5.4.3 Liveness. Recall that the step semantics of fair streams $\Longrightarrow_{\mathsf{F}}$ is a machine whose configurations are tuples $\langle t;\eta;p\rangle$, where $p \in \{1,2\}$ indicates the current mode of the computation. The behaviour of the different modes is captured by the following definition of the set $F(A,B,\alpha,\beta)$ of such pairs:

$$\begin{aligned} F(A,B,\alpha,\beta) = {} & \{\langle t;\eta;1\rangle \mid \langle t;\eta\rangle \in U(A, B\times \triangleright(B\ \mathcal{U}\ (A\times\triangleright\mathsf{Fair}(\mathsf{A},\mathsf{B}))), \alpha,\beta)\} \\ & \cup \{\langle t;\eta;2\rangle \mid \langle t;\eta\rangle \in U(B, A\times\triangleright\mathsf{Fair}(\mathsf{A},\mathsf{B}), \alpha,\beta)\} \end{aligned}$$

That is, if $p=1$, then t belongs semantically to an until type $A\ \mathcal{U}\ (B\times\triangleright\mathsf{Fair}'(\mathsf{B},\mathsf{A}))$, and otherwise t belongs to $B\ \mathcal{U}\ (A\times\triangleright\mathsf{Fair}(\mathsf{A},\mathsf{B}))$.

With this characterisation, we can formulate the essence of the liveness property for fair streams:

Lemma 5.10 (Liveness). *Given* $\langle t;\eta;p\rangle \in F(A,B,\alpha,\beta)$, *one of the following statements is true:*

(a) there are $t', \eta', \alpha' < \alpha$, *and* $v \in \mathcal{V}[\![A + B]\!](\eta', \overline{\emptyset}, \omega, \beta)$ *such that*

$$\langle t; \eta; p\rangle \overset{v}{\Longrightarrow}_{\mathsf{F}} \langle t'; \eta'; p\rangle \text{ and } \langle t'; \eta'; p\rangle \in F(A, B, \alpha', \beta') \text{ for all } \beta' < \beta.$$

(b) there are t', η' *and* $v \in \mathcal{V}[\![A + B]\!](\eta', \overline{\emptyset}, \omega, \beta)$ *such that*

$$\langle t; \eta; p\rangle \overset{v}{\Longrightarrow}_{\mathsf{F}} \langle t'; \eta'; 3 - p\rangle \text{ and } \langle t'; \eta'; 3 - p\rangle \in F(A, B, \omega, \beta') \text{ for all } \beta' < \beta.$$

The liveness result is now an easy consequence of the above lemma and the fundamental property:

Proof of Theorem 4.3. By Theorem 5.6 $\mathsf{out}\,(\mathsf{unbox}\, t) \in F(A, B, \omega, \omega + n)$ for any n. Using Lemma 5.10, we can then show that we can extend any finite reduction sequence

$$\langle \mathsf{out}\,(\mathsf{unbox}\, t); \eta_0; 1\rangle \overset{\mathsf{in}_{p_1} v_1}{\Longrightarrow}_{\mathsf{F}} \langle t_1; \eta_i; p_1\rangle \overset{\mathsf{in}_{p_2} v_2}{\Longrightarrow}_{\mathsf{F}} \langle t_2; \eta_0; p_2\rangle \overset{\mathsf{in}_{p_3} v_3}{\Longrightarrow}_{\mathsf{F}} \ldots \overset{\mathsf{in}_{p_n} v_n}{\Longrightarrow}_{\mathsf{F}} \langle t_n; \eta_n; p_n\rangle$$

with $\langle t_n; \eta_n; p_n\rangle \overset{\mathsf{in}_{p_{n+1}} v_{n+1}}{\Longrightarrow}_{\mathsf{F}} \langle t_{n+1}; \eta_{n+1}; p_{n+1}\rangle$ so that $\langle t_{n+1}; \eta_{n+1}; p_{n+1}\rangle \in F(A, B, \omega, \omega)$. Since $\Longrightarrow_{\mathsf{F}}$ is deterministic, this defines the desired infinite sequence of reductions. Moreover, since $\langle t_i; \eta_i; p_i\rangle \in F(A, B, \omega, \omega)$ for each i, and the index α decreases for every step of the form (a), we know that only finitely many reduction steps after $\langle t_i; \eta_i; p_i\rangle$ are of the form (a). Thus, there is a $j \geq i$ with $p_j \neq p_i$. In addition, given that A and B are value types, so is $A + B$, and we thus obtain by Lemma 5.7 that $\vdash \mathsf{in}_{p_i} v_i : A + B$ for all $i \geq 1$. □

5.4.4 Causality. We conclude by sketching the proofs for the corresponding operational properties for the reactive step semantics. The proof idea is the same but instead of setting heaps in $\overline{\eta}$ to the empty heap, we construct $\overline{\eta}$ so that it contains the input stream.

Recall that after each step of the (reactive) step semantics, the machine starts a new empty 'later' heap. Each heap comes with its own namespace. Let l_i be the location that is picked as the first location by the allocator after i steps of the (reactive) step semantics, i.e. $\mathsf{alloc}\,(\eta_i) = l_i$ where η_i is the empty heap after i steps. Given a value v and number $i \geq 0$, we write η_v^i for the heap $l_i \mapsto v :: l_{i+1}$. We further define the following set of heap sequences:

$$H(A, i) = \left\{\eta_{v_i}^i; \eta_{v_{i+1}}^{i+1}; \ldots \,\middle|\, \forall j \geq i.\, \vdash v_j : A\right\}$$

The locations l_i are used to feed input to the reactive step semantics. The following lemma shows that each l_i has the right semantic type:

Lemma 5.11. *For all* $i \geq 0, \vdash v : A$, *and* $\overline{\eta} \in H(A, i+1)$, *we have that* $l_i \in \mathcal{V}[\![\triangleright(\mathsf{Str}(A))]\!](\eta_v^i, \overline{\eta}, \omega, \beta)$.

The lemma is proved by a straightforward induction on β using Corollary 5.6.

We can now prove variants of Lemma 5.8, 5.9, and 5.10 for the reactive step semantics. For the reactive step semantics of streams $\overset{/}{\Longrightarrow}_{\mathsf{Str}}$, we define the corresponding set of machine configurations that are safe according to the logical relation as follows:

$$S^R(\overline{\eta}, B, \beta) = \left\{\langle t; \eta; l_i\rangle \,\middle|\, \overline{\eta} = \eta_v^i; \eta_w^{i+1}; \overline{\eta}' \wedge t \in \mathcal{T}[\![\mathsf{Str}(B)]\!]((\eta, \eta_v^i \checkmark \eta_w^{i+1}), \overline{\eta}', \omega, \beta)\right\}$$

This construction takes as additional parameter $\overline{\eta}$ drawn from $H(A, i)$ which represents future input. We can then formulate the corresponding productivity property as follows:

Lemma 5.12. *Given* $\langle t; \eta; l_i\rangle \in S^R((\eta_v^i; \overline{\eta}), B, \beta + 1)$ *and* $\overline{\eta} \in H(A, i+1)$, *then there are* $\langle t'; \eta'; l_{i+1}\rangle \in S^R(\overline{\eta}, B, \beta)$ *and* v' *such that*

$$\langle t; \eta; l_i\rangle \overset{v/v'}{\Longrightarrow}_{\mathsf{Str}} \langle t'; \eta'; l_{i+1}\rangle\,.$$

Moreover, if B *is a value type then* $\vdash v' : B$.

The constructions for $\overset{/}{\Longrightarrow}_{\mathcal{U}}$ and $\overset{/}{\Longrightarrow}_{\mathsf{F}}$ are analogous and we can then prove the causality property of the reactive step semantics.

Proof of Theorem 4.4. We give the proof for part (i) of the theorem. Part (ii) and (iii) follow by a similar adaptation of the proofs of Theorems 4.2 and 4.3, respectively. By Corollary 5.6, $\mathsf{unbox}\, t \in \mathcal{T}[\![\mathsf{Str}(A) \to \mathsf{Str}(B)]\!](\eta^0_{v_1} \checkmark \eta^1_{v_2}, \overline{\eta}, \omega, \beta)$ for all β and for $\overline{\eta} = \eta^2_{v_3}; \eta^3_{v_4}; \ldots$. By Lemma 5.11 $\mathsf{adv}\, l_0 \in \mathcal{T}[\![\mathsf{Str}(A)]\!](\eta^0_{v_1} \checkmark \eta^1_{v_2}, \overline{\eta}, \omega, \beta)$ and thus $\mathsf{unbox}\, t\ (\mathsf{adv}\, l_0) \in \mathcal{T}[\![\mathsf{Str}(B)]\!](\eta^0_{v_1} \checkmark \eta^1_{v_2}, \overline{\eta}, \omega, \beta)$. Hence, we have $\langle \mathsf{unbox}\, t\ (\mathsf{adv}\, l_0); \emptyset; l_0 \rangle \in U(\eta^0_{v_1}; \eta^1_{v_2}; \overline{\eta}, B, \beta)$ for any β. Similarly to the proof of Theorem 4.1, we can then use Lemma 5.12 to construct the desired infinite sequence of reduction steps. □

6 RELATED WORK

The work by Cave et al. [2014] mentioned in the introduction defines a language with a modal operator $\bigcirc$ as well as inductive and coinductive types, but no guarded fixed points. They define a family of reduction relations indexed by ordinals up to and including ω. The relations corresponding to finite ordinals describe reductions up to finitely many steps, and the one at ω describes global behaviour. They give an interpretation of types as predicates on values indexed by ordinals up to and including ω, and similarly to our interpretation of types, the interpretation of $\bigcirc A$ at ω refers to the interpretation of A also at ω. Using this they prove strong normalisation, and sketch proofs of causality, productivity and liveness, but they do not prove lack of space leaks as done here. The motivation for omitting the guarded fixed point operator is exactly the observation mentioned in the introduction that these equate inductive and coinductive types. Instead, programming with coinductive types like streams must be done by coiteration. The present paper shows how to refine the modal type system to combine the type system of LTL with the power of the fixed point operator, gaining simplicity in programming and productivity checking. The language of Cave et al. [2014] has more general inductive and coinductive types than Lively RaTT (but not general guarded recursive types), see discussion in section 7. The idea of transfinite step indexing as used both here and by Cave et al. [2014], has also been used to model countable non-determinism [Bizjak et al. 2014] and distinguishing between logical and and concrete steps in program verification [Svendsen et al. 2016].

Jeffrey [2012] and Jeltsch [2012] independently discovered the connection between FRP and LTL. Jeltsch [2012, 2013] studied a category theoretic common notion of models of LTL and FRP. Jeffrey [2012] defined a language for FRP as an abstraction of a model defined in a functional programming language. Signals are defined directly as time-dependent values and LTL types are defined by quantifying over time. While the native function space of the language contains all signal functions, a type of causal functions is definable in the language. In later work, Jeffrey [2014] extends modal FRP with heterogeneous stream types, i.e., streams of elements whose types are given by a stream of types, and use this to encode past-time LTL. Unlike the present work, neither Jeltsch, nor Jeffrey define an operational semantics of programs, and therefore prove no operational metatheoretical results.

To our knowledge, the first work to define a modal type theory for FRP with a guarded fixed point operator is that of Krishnaswami and Benton [2011]. This line of work also studies type systems for eliminating implicit space and time leaks. Krishnaswami et al. [2012] use linear types to statically bound the size of the dataflow graph generated by a reactive program, while Krishnaswami [2013] defines a simpler type system, but rules out space leaks using the techniques also used in the present paper. Bahr et al. [2019] recast this work in the setting of Simply RaTT, which unlike Krishnaswami [2013] uses Fitch style for programming with modal types, and extend these results by identifying and eliminating a type of time leaks stemming from fixed points.

The guarded fixed point operator was first suggested by Nakano [2000] and has since received much attention in logics for program verification because it can be used as a synthetic approach [Appel et al. 2007; Birkedal et al. 2011] to step-indexing [Appel and McAllester 2001]. Moreover, combining this with a notion of quantification over clocks [Atkey and McBride 2013] or a constant modality [Clouston et al. 2015] one can use guarded recursion to encode coinduction. Guarded recursion forms part of the foundation of the framework Iris [Jung et al. 2015] for higher-order concurrent separation logic in Coq, and a number of dependent type theories with guarded recursion have been defined [Bahr et al. 2017; Birkedal et al. 2019; Bizjak et al. 2016]. In the simply typed setting Guatto [2018] extends this with a notion of time warps. The combination of guarded recursion and higher inductive types [Univalent Foundations Program 2013] has also been used for modelling process calculi [Møgelberg and Veltri 2019; Veltri and Vezzosi 2020]. Although related to the modal FRP calculi, these systems are usually much more expressive, since space and time leaks are ignored in their design. For example, they all include an operation $A \to \triangleright A$ transporting data into the future, a known source of space leaks.

7 CONCLUSION AND FUTURE WORK

This paper shows how guarded fixed points can be combined with liveness properties in modal FRP. While properties such as termination, liveness and fairness are perhaps beyond the scope of properties traditionally expressed in simply typed programming languages, they could naturally occur as parts of program specifications in dependently typed languages and proof assistants. We therefore view Lively RaTT as a conceptual stepping stone towards a dependently typed language for reactive programming.

The results of this paper have been presented in the setting of functional reactive programming, but we expect that the ideas will be relevant also in the setting of guarded recursion as described in section 6. In these settings, the fact that inductive and coinductive types coincide means that termination cannot be expressed directly. This leads to limitations in the setting of program verification, e.g., when defining notions such as weak bisimulation for programs [Møgelberg and Paviotti 2019] and processes. We expect that the tools developed here can be used in this respect once this work has been adapted to guarded recursion and extended to dependent types.

Future work also includes extending Lively RaTT with general classes of inductive types. Note that as seen here one must distinguish between ordinary inductive types such as Nat and temporal ones such as the $\mathcal{U}$ types, where the recursion involves time steps and the recursors therefore need to be stable. The temporal inductive types should be defined as a class of strictly positive inductive types where the recursion variable appears under a $\bigcirc$, generalising the $\mathcal{U}$ types used here. One could likewise add a class of coinductive types in the ordinary sense, but the temporal coinductive types are subsumed by the guarded recursive types.

ACKNOWLEDGMENTS

This work was supported by a research grant (13156) from VILLUM FONDEN.

A ELABORATING SURFACE SYNTAX TO CORE CALCULUS

Throughout the paper we use syntactic sugar for writing Lively RaTT programs. In this appendix we give an overview how this surface syntax can be elaborated into the core calculus.

First, the use of pattern matching in function definitions is translated into (nested) case expressions in a standard way (including inlining shorthands such as $\mathsf{wait}_{\mathsf{Ev}}$). For example, the two clauses defining $bind_{\mathsf{Ev}}$ on page 8 are elaborated into the following single clause:

$$bind_{\mathsf{Ev}}\ f\ \sharp\ x = \text{case } x \text{ of } \mathsf{into}\ (\mathsf{in}_1\ a)\ .\ (\mathsf{unbox}\ f)\ a$$
$$\mathsf{into}\ (\mathsf{in}_2\ e)\ .\ \mathsf{wait}_{\mathsf{Ev}}\ (\mathsf{unbox}\ b \circledast e))$$

The resulting general case expressions with nested pattern matching are then transformed step-by-step into the elimination forms of the core calculus. For example the above case expression is transformed into:

$$\text{case } \mathsf{out}\ x \text{ of } \mathsf{in}_1\ a\ .\ (\mathsf{unbox}\ f)\ a$$
$$\mathsf{in}_2\ e\ .\ \mathsf{wait}_{\mathsf{Ev}}\ (\mathsf{unbox}\ b \circledast e))$$

Once these transformations are performed, all function definitions consist of a single clause (non-recursive functions and guarded recursive functions) or two clauses (functions defined by natural number or $\mathcal{U}$ recursion). We consider each of these cases in turn.

A guarded recursive function definition is of the form

$$f\ x_1 \dots x_n \sharp y_1 \dots y_m = t\ [f\ x_1 \dots x_n / r]$$

where f may not occur freely in t. This function definition is elaborated into the following term:

$$f = \lambda x_1 . \dots . \lambda x_n . \mathsf{fix}\, r . \lambda y_1 . \dots . \lambda y_m . t$$

Mutual guarded recursive function definitions define several functions simultaneously. We consider the case of two mutually guarded recursive functions, with the general case following in a similar manner:

$$f_1\ x_1 \dots x_n \sharp y_1 \dots y_m = t_1\ [f_1\ x_1 \dots x_n / r_1, f_2\ x_1 \dots x_n / r_2]$$
$$f_2\ x_1 \dots x_n \sharp z_l \dots z_l \quad = t_2\ [f_1\ x_1 \dots x_n / r_1, f_2\ x_1 \dots x_n / r_2]$$

where t_1 and t_2 do not contain free occurrences of f_1 or f_2. This definition is then transformed into

$$f = \mathsf{fix}\, r . \left\langle \lambda y_1 . \dots . \lambda y_m . t_1\ \left[\pi_1^{\Box}\ r / r_1, \pi_2^{\Box}\ r / r_2\right], \lambda z_1 . \dots . \lambda z_m . t_2\ \left[\pi_1^{\Box}\ r / r_1, \pi_2^{\Box}\ r / r_2\right] \right\rangle$$
$$f_1 = \lambda x_1 . \dots . \lambda x_n . \mathsf{box}(\pi_1(\mathsf{unbox}\, f))$$
$$f_2 = \lambda x_1 . \dots . \lambda x_n . \mathsf{box}(\pi_2(\mathsf{unbox}\, f))$$

where $\pi_i^{\Box} : \Box(\triangleright(A_1 \times A_2)) \to \Box(\triangleright A_i)$, for $i \in \{1, 2\}$, is defined by

$$\pi_i^{\Box} = \lambda x . \mathsf{box}(\mathsf{delay}(\pi_i(\mathsf{adv}(\mathsf{unbox}\, x))))$$

Functions defined by natural number recursion are of the form:

$$f\ 0 \qquad x_1 \dots x_n \sharp y_1 \dots y_m = s$$
$$f\ (\mathsf{suc}\, z)\ x_1 \dots x_n \sharp y_1 \dots y_m = t\ [f\ z / r]$$

where f may not occur freely in t. This definition is elaborated into the term

$$f = \lambda x . \mathsf{rec}_{\mathsf{Nat}}(\lambda x_1 . \dots . \lambda x_n . \mathsf{box}\ (\lambda y_1 . \dots . \lambda y_m . s), z\ r . \lambda x_1 . \dots . \lambda x_n . \mathsf{box}\ (\lambda y_1 . \dots . \lambda y_m . t), x)$$

In a recursive definition without $\sharp$, the transformation is the same but omits box.

Functions defined by $\mathcal{U}$ recursion are of the form:

$$f\ (\mathsf{now}\, x) \quad x_1 \dots x_n = s$$
$$f\ (\mathsf{wait}\, x\, t)\ x_1 \dots x_n = t\ [f\ y / r]$$

where f may not occur freely in t. This definition is elaborated into the term

$$f = \lambda z . \mathsf{rec}_{\mathcal{U}}(x . \lambda x_1 . \dots . \lambda x_n . s, x\ y\ r . \lambda x_1 . \dots . \lambda x_n . t, z)$$

REFERENCES

Andreas Abel and Brigitte Pientka. 2013. Wellfounded Recursion with Copatterns: A Unified Approach to Termination and Productivity. In *Proceedings ICFP 2013*. ACM, New York, NY, USA, 185–196. https://doi.org/10.1145/2500365.2500591

Andreas Abel, Andrea Vezzosi, and Théo Winterhalter. 2017. Normalization by evaluation for sized dependent types. *PACMPL* 1, ICFP (2017), 33:1–33:30. https://doi.org/10.1145/3110277

Andrew W. Appel and David McAllester. 2001. An Indexed Model of Recursive Types for Foundational Proof-carrying Code. *ACM Trans. Program. Lang. Syst.* 23, 5 (Sept. 2001), 657–683. https://doi.org/10.1145/504709.504712 00283.

Andrew W Appel, Paul-André Mellies, Christopher D Richards, and Jérôme Vouillon. 2007. A very modal model of a modern, major, general type system. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, New York, NY, USA, 109–122. https://doi.org/10.1145/1190215.1190235

Robert Atkey and Conor McBride. 2013. Productive coprogramming with guarded recursion. *ACM SIGPLAN Notices* 48, 9 (2013), 197–208. https://doi.org/10.1145/2500365.2500597

Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. 2017. The clocks are ticking: No more delays!. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. IEEE Computer Society, Washington, DC, USA, 1–12. https://doi.org/10.1109/LICS.2017.8005097

Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. 2019. Simply RaTT: a fitch-style modal calculus for reactive programming without space leaks. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–27. https://doi.org/10.1145/3341713

Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. 2019. Guarded Cubical Type Theory. *Journal of Automated Reasoning* 63, 2 (01 Aug 2019), 211–253. https://doi.org/10.1007/s10817-018-9471-7

Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. 2011. First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. In *In Proc. of LICS*. IEEE Computer Society, Washington, DC, USA, 55–64. https://doi.org/10.2168/LMCS-8(4:1)2012

Aleš Bizjak, Lars Birkedal, and Marino Miculan. 2014. A Model of Countable Nondeterminism in Guarded Type Theory. In *Rewriting and Typed Lambda Calculi*, Gilles Dowek (Ed.). Springer International Publishing, Cham, 108–123. https://doi.org/10.1007/978-3-319-08918-8_8

Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal. 2016. Guarded Dependent Type Theory with Coinductive Types. In *Foundations of Software Science and Computation Structures*, Bart Jacobs and Christof Löding (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 20–35. https://doi.org/10.1007/978-3-662-49630-5_2

Andrew Cave, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka. 2014. Fair Reactive Programming. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. ACM, San Diego, California, USA, 361–372. https://doi.org/10.1145/2535838.2535881

Ranald Clouston. 2018. Fitch-Style Modal Lambda Calculi. In *Foundations of Software Science and Computation Structures*, Christel Baier and Ugo Dal Lago (Eds.). Springer International Publishing, Cham, 258–275. https://doi.org/10.1007/978-3-319-89366-2_14

Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. 2015. Programming and Reasoning with Guarded Recursion for Coinductive Types. In *Foundations of Software Science and Computation Structures*, Andrew Pitts (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 407–421. https://doi.org/10.1007/978-3-662-46678-0_26

Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. 2018. Modal Dependent Type Theory and Dependent Right Adjoints. *CoRR* abs/1804.05236 (2018), 1–21. arXiv:1804.05236 http://arxiv.org/abs/1804.05236

Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*. ACM, New York, NY, USA, 263–273. https://doi.org/10.1145/258948.258973

Frederic Benton Fitch. 1952. *Symbolic logic, an introduction*. Ronald Press Co., New York, NY, USA. https://doi.org/10.2307/2266614

Adrien Guatto. 2018. A Generalized Modality for Recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS '18)*. Association for Computing Machinery, New York, NY, USA, 482–491. https://doi.org/10.1145/3209108.3209148

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. Association for Computing Machinery, New York, NY, USA, 410–423. https://doi.org/10.1145/237721.240882

Alan Jeffrey. 2012. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*, Koen Claessen and Nikhil Swamy (Eds.). ACM, Philadelphia, PA, USA, 49–60. https://doi.org/10.1145/2103776.2103783

Alan Jeffrey. 2014. Functional Reactive Types. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS) (CSL-LICS '14)*. ACM, New York, NY, USA, Article 54, 9 pages. https://doi.org/10.1145/2603088.2603106

Wolfgang Jeltsch. 2012. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electronic Notes in Theoretical Computer Science* 286 (2012), 229–242. https://doi.org/10.1016/j.entcs.2012.08.015

Wolfgang Jeltsch. 2013. Temporal Logic with "Until", Functional Reactive Programming with Processes, and Concrete Process Categories. In *Proceedings of the 7th Workshop on Programming Languages Meets Program Verification (PLPV '13)*. ACM, New York, NY, USA, 69–78. https://doi.org/10.1145/2428116.2428128

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. *ACM SIGPLAN Notices* 50, 1 (2015), 637–650. https://doi.org/10.1145/2775051.2676980

Neelakantan R. Krishnaswami. 2013. Higher-order Functional Reactive Programming Without Spacetime Leaks. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP '13)*. ACM, Boston, Massachusetts, USA, 221–232. https://doi.org/10.1145/2500365.2500588

Neelakantan R. Krishnaswami and Nick Benton. 2011. Ultrametric Semantics of Reactive Programs. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*. IEEE Computer Society, Washington, DC, USA, 257–266. https://doi.org/10.1109/LICS.2011.38

Neelakantan R. Krishnaswami, Nick Benton, and Jan Hoffmann. 2012. Higher-order functional reactive programming in bounded space. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, John Field and Michael Hicks (Eds.). ACM, Philadelphia, PA, USA, 45–58. https://doi.org/10.1145/2103656.2103665

Saunders MacLane and Ieke Moerdijk. 2012. *Sheaves in geometry and logic: A first introduction to topos theory*. Springer Science & Business Media, New York, NY, USA. https://doi.org/10.1007/978-1-4612-0927-0

Rasmus E Møgelberg and Marco Paviotti. 2019. Denotational semantics of recursive types in synthetic guarded domain theory. *Mathematical Structures in Computer Science* 29, 3 (2019), 465–510. https://doi.org/10.1017/S0960129518000087

Rasmus Ejlers Møgelberg and Niccolò Veltri. 2019. Bisimulation as path type for guarded recursive types. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29. https://doi.org/10.1145/3290317

Hiroshi Nakano. 2000. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*. IEEE Computer Society, Washington, DC, USA, 255–266. https://doi.org/10.1109/LICS.2000.855774

Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*. ACM, New York, NY, USA, 51–64. https://doi.org/10.1145/581690.581695

Amir Pnueli. 1977. The Temporal Logic of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science (SFCS '77)*. IEEE Computer Society, USA, 46–57. https://doi.org/10.1109/SFCS.1977.32

Jorge Luis Sacchini. 2013. Type-Based Productivity of Stream Definitions in the Calculus of Constructions. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*. IEEE, New Orleans, LA, USA, 233–242. https://doi.org/10.1109/LICS.2013.29

Kasper Svendsen, Filip Sieczkowski, and Lars Birkedal. 2016. Transfinite Step-Indexing: Decoupling Concrete and Logical Steps. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 727–751. https://doi.org/10.1007/978-3-662-49498-1_28

The Univalent Foundations Program. 2013. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study.

Niccolò Veltri and Andrea Vezzosi. 2020. Formalizing π-Calculus in Guarded Cubical Agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2020)*. Association for Computing Machinery, New York, NY, USA, 270–283. https://doi.org/10.1145/3372885.3373814

Chapter 5

Paper 3: Adjoint Reactive GUI

Adjoint Reactive GUI Programming

Christian Uldal Graulund[1], Dmitrij Szamozvancev[2], and Neel Krishnaswami[2]

[1] IT University of Copenhagen, 2300 Copenhagen, DK cgra@itu.dk
[2] University of Cambridge, Cambridge CB3 0FD, UK nk480@cl.cam.ac.uk,ds709@cl.cam.ac.uk

Abstract. Most interaction with a computer is done via a graphical user interface. Traditionally, these are implemented in an imperative fashion using shared mutable state and callbacks. This is efficient, but is also difficult to reason about and error prone. Functional Reactive Programming (FRP) provides an elegant alternative which allows GUIs to be designed in a declarative fashion. However, most FRP languages are synchronous and continually check for new data. This means that an FRP-style GUI will "wake up" on each program cycle. This is problematic for applications like text editors and browsers, where often nothing happens for extended periods of time, and we want the implementation to sleep until new data arrives. In this paper, we present an *asynchronous* FRP language for designing GUIs called $\lambda_{\mathsf{Widget}}$. Our language provides a novel semantics for widgets, the building block of GUIs, which offers both a natural Curry–Howard logical interpretation and an efficient implementation strategy.

Keywords: Linear Types · Reactive Programming · Asynchronous Programming · Graphical User Interfaces

1 Introduction

Many programs, like compilers, can be thought of as functions – they take a single input (a source file) and then produce an output (such as a type error message). Other programs, like embedded controllers, video games, and integrated development environments (IDEs), engage in a dialogue with their environment: they receive an input, produce an output, and then wait for a new input that depends on the prior input, and produce a new output which is in turn potentially based on the whole history of prior inputs.

Our usual techniques for programming interactive applications are often very confusing, because the different parts of the program are not written to interact via structured control flow (i.e., by passing and return values from functions, or iterating over data in loops). Instead, they communicate indirectly, by registering state-manipulating callbacks with one another, which are then implicitly invoked by an event loop. This makes program reasoning very challenging, since each of these features – aliased mutable state, higher-order functions, and concurrency – represents a serious obstacle on its own, and interactive programs rely upon their *combination*.

This challenge has led to a great deal of work on better abstractions for programming reactive systems. Two of the main lines of work on this problem are *synchronous dataflow* and *functional reactive programming*. The synchronous dataflow languages, like Esterel [5], Lustre [9], and Lucid Synchrone [28], feature a programming model inspired by Kahn networks. Programs are networks of stream-processing nodes which communicate with each other, each node consuming and producing a fixed number of primitive values at each clock tick. The first-order nature of these languages makes them strongly analysable, which lets them offer powerful guarantees on space and time usage. This means they see substantial use in embedded and safety-critical contexts.

Functional reactive programming, introduced by Elliott and Hudak [13], also uses time-indexed values, dubbed signals, rather than mutable state as its basic primitive. However, FRP differs from synchronous dataflow by sacrificing static analysability in favour of a much richer programming model. Signals are true first-class values, and can be used freely, including in higher-order functions and signal-valued signals. This permits writing programs with a dynamically-varying dataflow network, which simplifies writing programs (such as GUIs) in which the available signals can change as the program executes. Over the past decade, a long line of work has refined FRP via the Curry–Howard correspondence [21,18,17,19,20,10,1]. This approach views functional reactive programs as the programming counterpart for proofs of formulas in linear temporal logic [27], and has enabled the design of calculi which can rule out spacetime leaks [20] or can enforce temporal safety and liveness properties [10].

However, both synchronous dataflow and FRP (in both original and modal flavours) have a *synchronous* (or "pull") model of time – time passes in ticks, and the program wakes up on every tick to do a little bit more computation. This is suitable for applications in which something new happens at every time step (e.g., video games), but many GUI programs like text editors and spreadsheets spend most of their time doing nothing. That is, even at each event, most of the program will continue doing nothing, and we only want to wake up a component when an event directly relevant to it occurs. This is important both from a performance point of view, as well as for saving energy (and extending battery life). Because of this need, most GUI programs continue to be written in the traditional callbacks-on-mutable-state style.

In this paper, we give a reactive programming language whose type system both has a very straightforward logical reading, and which can give natural types to stateful widgets and the event-based programming model they encourage. We also derive a denotational semantics of the language, by first working out a semantics of widgets in terms of the operations that can be performed upon them and the behaviour they should exhibit. Then, we find the categorical setting in which the widget semantics should live, and by studying the structure this setup has, we are able to interpret all of the other types of the programming language.

1.1 Contributions

The contributions of this paper are as follows:

- We give a descriptive semantics for widgets in GUI programming, and show that this semantics correctly models a variety of expected behaviours. For example, our semantics shows that a widget which is periodically re-set to the colour red is different from a widget that was only persistently set to the colour red at the first timestep. Our semantic model can show that as long as neither one is updated, they look the same, but that they differ if they are ever set to blue – the first will return to red at reset time, and the second will remain blue.
- From this semantics, we find a categorical model within which the widget semantics naturally fits. This model is a Kripke–Joyal presheaf semantics, which is morally a "proof-relevant" Kripke model of temporal logic.
- We give a concrete calculus for event-based reactive programming, which can be implemented in terms of the standard primitives for modern GUI programming, scene graphs (or DOM) which are updated via callbacks invoked upon events. We then show that our model can soundly interpret the types of our calculus in an entirely standard way, showing that the types of our reactive programming language can be interpreted as time-varying sets.

– Furthermore, this calculus has an entirely standard logical reading in terms of the Curry–Howard correspondence. It is a "linear temporal linear logic", with the linear part of the language corresponding to the Benton–Wadler [3] LNL calculus for linear logic, and the temporal part of the language corresponding to S4.3 linear temporal logic. We also give a proof term for the $S_t4.3$ axiom enforcing the linearity of time, and show that it corresponds to the `select` primitive of concurrent programming.

2 The Language

In this section we give an informal presentation of $\lambda_{\mathsf{Widget}}$ through the API of the Widget type. This API mirrors how one would work with a GUI at the browser level. An important feature of a well-designed GUI is that it should not do anything when not in use. In particular, it should not check for new inputs in each program cycle (*pull*-based reactive programming), but rather sleep until new data arrives (*push*-based reactive programming). Many FRP languages are *synchronous* languages and have some internal notion of a timestep. These languages are mostly pull-based, whereas more traditional imperative reactive languages are push-based. The former have clear semantics and are easy to reason about, the latter have efficient implementations. In $\lambda_{\mathsf{Widget}}$ we would like to combine these aspects and get a language that is easy to reason about with an efficient implementation.

In general, we think of a widget as a *state through time*, i.e., at each timestep, the widget is in some state which is presented to the user. The widget is modified by *commands*, which can update the state. To program with widgets, the programmer applies commands at various times.

The proper type system for a language of widgets should thus be a system with both state and time. If we consider what a *logic* for widgets should be, there are two obvious choices. A logic for state is linear logic [14], and a logic for time is linear temporal logic [27]. The combination of these two is the correct setting for a language of widgets, and, going through Curry–Howard, the corresponding type theory is a linear, linear temporal type theory.

2.1 Widget API

To work with widgets, we define a API which mirrors how one would work with a browser level GUI:

$$
\begin{array}{ll}
\mathsf{newWidget} & : \mathsf{I} \multimap \exists\,(i : \mathsf{Id}), \mathsf{Widget}\ i \\
\mathsf{dropWidget} & : \forall\,(i : \mathsf{Id}), \mathsf{Widget}\ i \multimap \mathsf{I} \\
\mathsf{setColor} & : \forall\,(i : \mathsf{Id}), \mathsf{F\ Color} \otimes \mathsf{Widget}\ i \multimap \mathsf{Widget}\ i \\
\mathsf{onClick} & : \forall\,(i : \mathsf{Id}), \mathsf{Widget}\ i \multimap \mathsf{Widget}\ i \otimes \Diamond\,\mathsf{I} \\
\mathsf{onKeypress} & : \forall\,(i : \mathsf{Id}), \mathsf{Widget}\ i \multimap \mathsf{Widget}\ i \otimes \Diamond\,(\mathsf{F\ Char}) \\
\mathsf{out} & : \Diamond\,A \multimap \exists\,(n : \mathsf{Time}), A \mathbin{@} n \\
\mathsf{into} & : \exists\,(n : \mathsf{Time}), A \mathbin{@} n \multimap \Diamond\,A \\
\mathsf{split} & : \forall\,(i : \mathsf{Id})\ (t : \mathsf{Time}), \mathsf{Widget}\ i \multimap \mathsf{Prefix}\ i\ t \otimes (\mathsf{Widget}\ i) \mathbin{@} t \\
\mathsf{join} & : \forall\,(i : \mathsf{Id})\ (t : \mathsf{Time}), \mathsf{Prefix}\ i\ t \otimes (\mathsf{Widget}\ i) \mathbin{@} t \multimap \mathsf{Widget}\ i
\end{array}
$$

The first two commands creates and deletes widgets, respectively. The $\multimap$ should be understood as *state passing*. We read the type of newWidget as "consuming no state, produce a new identifier index and a widget with that identifier index". The identifier indices are used to ensure the correct

behavior when using the split and join commands explained below. The existential quantification describes the *non-deterministic* creation of an identifier index. The use of non-determinism is crucial in our language and will be explaining in further detail in section 4. Since $\lambda_{\mathsf{Widget}}$ has a linear type system, we need an explicit construction to delete state. For widgets, this is dropWidget. The type is read as "for any identifier index, consume a widget with that identifier index and produce nothing".

The first command that modifies the state of a widget is setColor. Here we see the adjoint nature of the calculus with F Color. A color is itself *not* a linear thing, and as such, to use it in the linear setting, we apply F, which moves from the non-linear (Cartesian) fragment and into the linear fragment. The second new thing is the linear product $\otimes$. This differs from the regular non-linear product in that we do not have projection maps. Again, because of the linearity of our language, we cannot just discard state. We can now read the type of setColor as "Given a color and a identified widget, consume both and produce a new widget". The produced widget is the same as the consumed widget, but with the color attribute updated.

The next two commands, onClick and onKeypress, are roughly similar. Both register a handle on the widget, for a mouse click and a key press, respectively. Here we see the first use of the $\Diamond$ modality, which represents an *event*. The type $\Diamond A$ represents that *at some point in the future* we will receive something of type A. Importantly, because of the asynchronous nature of $\lambda_{\mathsf{Widget}}$, we do not know *when* it happens. We can then read the type of onClick as "Consuming an identified widget, produce an updated widget together with a mouse click event". The same holds for the type of onKeypress except a key press event is produced.

The two commands out and into allows us to work with events in a more precise way. Given an event, we can use out to "unfold" it into an existential. The @ connective describes a type that is only available at a certain timestep, i.e., $A \mathbin{@} n$ means "at the timestep n, a term of type A will be available". The into commands is the reverse of out and turns an existential and an @ into an event.

So far, we have only applied commands to a widget in the current timestep, but to program appropriately with widgets, we should be able to react to events and apply commands "in the future". This is exactly what the split and join commands allows us to do. The type of split is read as "Given any time step and any identified widget, split the widget into all the states *before* that time and the widget *at* that time". We denote the collection of states before a given time a *prefix* and give it the type Prefix. Given the state of the widget at a given timestep, we can now apply commands *at that timestep*. Note that both the prefix and the widget is indexed by the same identifier index. This is to ensure that when we use join, we combined the correct prefix and future.

2.2 Widget Programming

To see the API in action, we now proceed with several examples of widget programming. For each example, we will add a comment on each line with the type of variables, and then explain the example in text afterwards.

One of the simplest things we can do with a widget is to perform some action when the widget is clicked. In the following example, we register a handler for mouse clicks, and then we use the click event to change the color of the widget to red at the time of the click. To do this, we use the out map to get the time of the event, then we split the widget and apply setColor at that point in the future.

```
1 turnRedOnClick : ∀ (i : Id), Widget i ⊸ Widget i
2 turnRedOnClick i w₀ =
```

```
3   let (w_1, c_0)          = onClick i w_0 in                  -- w_1 : Widget i, c_0 : ◇I.
4   let unpack (x, c_1)     = out c_0 in                        -- x : Time, c_1 : I @ x.
5   let c_2 @ x             = c_1 in                            -- c_2 : I at the time x.
6   let ⟨⟩ @ x               = c_2 in
7   let (p, w_2)            = split i x w_2 in                  -- p : Prefix i x, w_2 : Widget i @ x.
8   let w_3 @ x             = w_2 in                            -- w_3 : Widget i at the time x.
9   let w_4                 = (setColor (F Red) w_3) @ x in     -- w_4 : Widget i @ x.
10  join i x (p, w_4)
```

To see why this type checks, we go through the example line by line. In line 3, we register a handle for a mouse click on the widget. In line 4, we turn the click event into an existential. In line 5, we get c_2 which is a binding that is only available at the timestep x. Since we only need the *time* of the click, we discharge the click itself in line 6. In line 7 and 8, we split the widget using the timestep x and bind w_3 to the state of the widget at that timestep. In line 9, we change the color of the widget to red at x and in line 10 we recompose the widget.

In general, we will allow pattern matching in eliminations and since widget identity indices can always be inferred, we will omit them. In this style, the above example become:

```
1  turnRedOnClick : ∀ (i : Id), Widget i ⊸ Widget i
2  turnRedOnClick w_0 =
3    let (w_1, c_0)            = onClick w_1 in    -- w_1 : Widget i, c_0 : ◇I.
4    let unpack (x, ⟨⟩ @ x)    = out c_0 in        -- x : Time.
5    let (p, w_2 @ x)          = split x w_1 in    -- p : Prefix i x, w_2 : Widget i at the time x.
6    join x (p, (setColor (F Red) w_2) @ x)
```

We will use the same sugared style throughout the rest of the examples.

The above example turns a widget red exactly at the time of the mouse click, but will not do anything with successive clicks. To also handle further mouse clicks, we must register an event handler *recursively*. This is a simple modification of the previous code:

```
1  keepTurningRed : ∀ (i : Id), Widget i ⊸ Widget i
2  keepTurningRed w_0 =
3    let (w_1, c_0)            = onClick w_1 in    -- w_1 : Widget i, c_0 : ◇I.
4    let unpack (x, ⟨⟩ @ x)    = out c_0 in        -- x : Time.
5    let (p, w_2 @ x)          = split x w_1 in    -- p : Prefix i x, w_2 : Widget i at the time x.
6    join (p, (setColor (F Red) (keepTurningRed w_2) @ x))
```

By calling itself recursively, this function will make sure a widget will always turn red on a mouse click.

To understand the difference between two above examples, consider the following code

```
1  exampleWidget : ∀ (i : Id), Widget i ⊸ Widget i
2  exampleWidget w = turnBlueOnClick (keepTurningRed w)
```

where w is some widget and *turnBlueOnClick* is the obvious modification of the above code. On the first click, the widget will turn blue, on the second click it will turn red and on any subsequent click, it will keep turning red, i.e., stay red unless further modified.

When working with widgets, we will often register multiple handlers on a single widget. For example, a widget should have one behavior for a click and another behavior for a key press.

To choose between two events, we use the select construction. This construction is central to our language and how to think about a push-based reactive language.

Given two events, $t_1 : \Diamond A, t_2 : \Diamond B$, there are three possible behaviors: Either t_1 returns first, and we wait for t_2 or t_2 returns first and we wait for t_1 or they return at the same time. In general, we want to select between n events, but if we need to handle all possible cases, this will give 2^n cases, so to keep the syntax linear in size, we will omit the last case. In the case events *actually* return at the same time, we do a non-deterministic choice between them. The syntax for select is

$$\mathsf{select}\ (t_1\ \mathsf{as}\ x \mapsto t_1' \mid t_2\ \mathsf{as}\ y \mapsto t_2')$$

where $x : A, y : B, t_1' : A \multimap \Diamond B \multimap \Diamond C$ and $t_2' : B \multimap \Diamond A \multimap \Diamond C$. The second important thing to understand when working with select is that given we are working with events, we do not actually know at which timestep the events will trigger, and hence, we do not know what the (linear) context contains. Thus, when using select, we will *only* know either $a : A, t_2 : \Diamond B$ or $t_1 : \Diamond A, b : B$. We can think of the select rule a *case-expression* that must respect time.

In the following example, we register two handlers, one for clicks and one for key presses, and change the color of the widget based on which returns first.

```
1  widgetSelect : ∀ (i : Id), Widget i ⊸ Widget i
2  widgetSelect w₀ =
3    let (w₁, c)              = onClick w₀ in        -- w₁ : Widget i, c : ◇I.
4    let (w₂, k)              = onKeypress w₁ in     -- w₂ : Widget i, k : ◇(F char).
5    let col                  =                      -- col : ◇(F Color)
6      select ( c as x → let ⟨⟩ = x in                -- x : I, k : ◇(F Color).
7                        evt (F Red)
8             | k as y → let F k' = y in              -- y : F char, c : ◇I
9                        evt (F Blue))
10   let unpack (x, col' @ x) = out col in           -- x : Time, col' : F Color at the time x.
11   let (p, w₃ @ x)          = split x w₂ in        -- p : Prefix i x, w₃ : Widget i at the time x.
12   join (p, (setColor col' w₃) @ x)
```

In line 3 and 4, we register the two handlers. In line 5-9, we use the select construction. In the first case, the click happens first and we return the color red. In the second case, the key press happens first and we return the color blue. In both cases, because of the linear nature of the language, we need to do a let-binding to discharge the unit and the char, respectively. In line 10, we turn the color event into an existential. In line 11, we use the timestep of the color event to split the widget, and in line 12, we change the color of the widget at that time and recompose it.

To see how $\lambda_{\mathsf{Widget}}$ differs from more traditional synchronous FRP languages, we will examine how to encode a kind of streams. Since our language is *asynchronous*, the stream type must be encoded as

$$\mathsf{Str}\ A := \nu\alpha.\Diamond(A \otimes \alpha)$$

This asynchronous stream will *at some point in the future* give a head and a tail. We do not know when the first element of the stream will arrive, and after each element of the stream is produced, we will wait an indeterminate amount of time for the next element. The reason why the stream type in $\lambda_{\mathsf{Widget}}$ must be like this is essentially that we want a *push-based* language, i.e., we do not want to wake up and check for new data in each program cycle. Instead, the program should sleep until new data arrives.

To show the difference between the asynchronous stream and the more traditional synchronous stream, we will look at some examples. With a traditional stream, a standard operation is zipping two streams: that is, given $\mathsf{Str}\ A$ and $\mathsf{Str}\ B$, we can produce $\mathsf{Str}\ A \times B$, which should be the element-wise pairing of the two streams. It should be clear that this is not possible for our asynchronous streams. Given two streams, we can wait until the first stream produces an element, but the second stream may only produce an element after a long period of time. Hence, we would need to buffer the first element, which is not supported in general. Remember, when using select, we can not use any already defined linear variables, since we do not know if they will be available in the future.

Rather than zipping stream, we can instead do a kind of *interleaving* as shown below. We use fold and unfold to denote the folding and unfolding of the fixpoint.

```
1 interleave : Str A ⊸ Str B ⊸ Str (A ⊕ B)
2 interleave xs ys = fold (
3   select ( unfold xs as xs' → let (x, xs'') = xs' in    -- xs' : A ⊗ ◇Str A, x : A, xs'' : ◇Str A
4                               evt (inl x, interleave xs' ys)
5          | unfold ys as ys' → let (y, ys'') = ys' in    -- ys' : B ⊗ ◇Str B, y : B, ys'' : ◇Str A
6                               evt (inr y, interleave xs ys')))
```

Here, we use select to choose between which stream returns first, and then we let that element be the first element of the new stream.

On the other hand, some of the traditional FRP functions on streams can be translated. For instance, we can map of function over a stream, given that *it is available at each step in time*:

```
1 map : F (G (A ⊸ B)) ⊸ Str A ⊸ Str B
2 map f0 xs =
3   let F f1          = f0 in          -- f1 : G(A ⊸ B) in Cartesian context
4   let (y, (x, xs') @ y) = out (unfold xs) in   -- y : Time and x : A, xs', ◇Str A at the time y.
5   fold (evt ((runG f1) x, map f0 xs'))
```

The type $\mathsf{F}(\mathsf{G}(A \multimap B))$ is read as a linear function with no free variables that can be used in a non-linear fashion, i.e., duplicated. This restriction to such "globally available functions" is reminiscent of the "box" modality in Bahr et al. [1] and Krishnaswami [20], and the F and G construction can be understood as decomposing the box modality into two separate steps. This relationship will be made precise in the logical interpretation of $\lambda_{\mathsf{Widget}}$ in section 3

As a final example, we will show how to dynamically update the GUI, i.e., how to add new widgets on the fly. Before we can give the example, we need to extend our widget API, to allow composition of widgets. To that end, we add the $\mathsf{vAttach}$ command to our API.

$$\mathsf{vAttach} : \forall (i, j : \mathsf{Id}), \mathsf{Widget}\ i \multimap \mathsf{Widget}\ j \multimap \mathsf{Widget}\ i$$

This command should be understood as an abstract version the div tag in HTML. In the following example, we think of the widget as a simple button that when clicked, will create a new button. When *any* of the buttons gets clicked, a new button gets attached.

```
1 buttonStack : ∀ i, Widget i ⊸ Widget i
2 buttonStack w0 =
3   let (w1, c)      = onClick w0 in
4   let (x, ⟨⟩ @ x)  = out e in
```

```
5  let (p, w2 @ x) = split x w1 in
6  let w3          = (let (y, w) = newWidget ⟨⟩ in
7                     vAttach w2 (buttonStack w)) @ x in
8  join (p, w3)
```

The important step here is in line 6 and 7. Here the new button is attached at the time of the mouse click, and *buttonStack* is called recursively on the newly created button.

3 Formal Calculus

This sections gives the formal rules, the meta-theory and the logical interpretation of $\lambda_{\mathsf{Widget}}$. Briefly, the language is an mixed linear-non-linear adjoint calculus in the style of Benton–Wadler [4,3]. The non-linear fragment, also called Cartesian in the following, is a minimal simply typed lambda calculus whereas the linear fragment contains several non-standard judgments used for widget programming.

3.1 Contexts and Typing Judgments

We have three separate typing judgments: one for indices, one for Cartesian (non-linear) terms, and one for linear terms. These are distinguished by a subscript on the turnstile, i for indices, c for Cartesian terms and l for linear terms. These depend on different contexts. The index judgment depends only on a index context, whereas the Cartesian and linear judgments depends on both an index and a linear and/or a Cartesian context. The rules for context formation is given in Figure 1. These are mostly standard except for the dependence on a previously defined context and the fact that the linear context contains variables of the form $a :_\tau A$, i.e., temporal variables. The judgment $a :_\tau A$ is read as "a has the type A at the timestep τ". In the linear setting we will write $a : A$ instead of $a :_0 A$, i.e., a judgment in the current timestep.

$$\text{Indices:}\quad \frac{}{\vdash_i \cdot} \qquad \frac{\vdash_i \Theta \qquad s \notin \mathsf{dom}(\Theta) \qquad \sigma \in \{\mathsf{Id}, \mathsf{Time}\}}{\vdash_i \Theta, s : \sigma}$$

$$\text{Cartesian:}\quad \frac{}{\cdot \vdash_c} \qquad \frac{\Theta \vdash_c \Gamma \qquad x \notin \mathsf{dom}(\Gamma) \qquad \Theta \vdash_c X}{\Theta \vdash_c \Gamma, x : X}$$

$$\text{Linear:}\quad \frac{}{\cdot \vdash_l} \qquad \frac{\Theta \vdash_l \Delta \qquad x \notin \mathsf{dom}(\Delta) \qquad \Theta \vdash_l A \qquad \Theta \vdash_i \tau : \mathsf{Time}}{\Theta \vdash_l \Delta, a :_\tau A}$$

Fig. 1. Context Formation

The index judgment describes how to introduce indices. The typing rules are given in Figure 2. The judgment $\Theta \vdash_i \tau : \sigma$ contains a single context, Θ, for index variables. There are only two sorts of indices, identifiers and timesteps.

The Cartesian judgment describes the Cartesian, or non-linear, fragment. The typing rules are given in Figure 3. This is a minimal simply typed lambda calculus with the addition of the G

Index Judgments:

$$\frac{\tau \in \mathsf{Time}}{\Theta \vdash_i \tau : \mathsf{Time}}\ \textsc{Time} \qquad \frac{\iota \in \mathsf{Id}}{\Theta \vdash_i \iota : \mathsf{Id}}\ \textsc{Id} \qquad \frac{i : \sigma \in \Theta}{\Theta \vdash_i i : \sigma}\ \textsc{Var}$$

Fig. 2. Index Typing rules

Cartesian Judgments:

$$\frac{}{\Theta; \Gamma \vdash_c \star : 1}\ (1\text{-I}) \qquad \frac{(x : X) \in \Gamma}{\Theta; \Gamma \vdash_c x : X}\ (\textsc{Var}) \qquad \frac{\Theta; \Gamma, x : X \vdash_c e : Y}{\Theta; \Gamma \vdash_c \lambda x.e : X \to Y}\ (\to\text{-I})$$

$$\frac{\Theta; \Gamma \vdash_c e_1 : X \to Y \qquad \Theta; \Gamma \vdash_c e_2 : X}{\Theta; \Gamma \vdash_c e_1 e_2 : Y}\ (\to\text{-E}) \qquad \frac{\Theta; \Gamma; \cdot \vdash_l t : A}{\Theta; \Gamma \vdash_c \mathsf{G}\ t : \mathsf{G}\ A}\ (\mathsf{G}\text{-I})$$

Fig. 3. Cartesian Typing rules

type, used for moving between the linear and Cartesian fragment, and explained further below. The judgment $\Theta; \Gamma \vdash_c t : A$ has two contexts; Θ for indices and Γ for Cartesian variables.

The linear fragment is most of the language, and the typing rules are given in Figure 4. The judgment is done w.r.t three contexts, Θ for index variables, Γ for Cartesian variables and Δ for linear variables. Many of the rules are standard for a linear calculus, except for the presence of the additional contexts. We will not describe the standard rules any further.

The first non-standard rule is for $\diamond$. The introduction and elimination rules follow from the fact that $\diamond$ is a non-strong monad. More interesting is the select rule. Here we see the formal rule corresponding to the informal explanation in section 2. The important thing here is that we can not use any previously defined linear variable when typing t_1' and t_2', since we do not actually know *when* the typing happens. Note, we can see the select rule as a binary version of the $\diamond$ let-binding. This could additionally be extended to a n-ary version, but we do not do this in our core calculus. The rules for $A\,@\,\tau$ shows how to move between the judgment $t : A\,@\,\tau$ and $t :_\tau A$. That is, moving from knowing in the current timestep that t will have the type A at time τ and knowing at time τ that t has type A. The (F -I), (F -E), (G -I) and (G -E) rules show the adjoint structure of the language. The (G -I) rule takes a closed linear term of type A and gives it the Cartesian type $\mathsf{G}\ A$. Note, because it has no free linear variables, it is safe to duplicate. The (G -E) rule lets us get an A without needing any linear resources. Conversely, the (F -I) rule embeds a intuitionistic term into the linear fragment and the (F -E) rule binds an intuitionistic variable to let us freely use the value. The quantification rules ($\exists$ and $\forall$) should also be familiar, except for the additional contexts. The (Delay) rule shows what happens when we actually *know* the timestep. The important part is $\Delta' = \Delta \downarrow^\tau$ which means two things. One, all the variables in Δ are on the form $a :_\tau A$, i.e., judgments at time τ and two, we shift Δ into the future such that all the variables of Δ' is of the form $a : A$. The way to understand this is, if all the variables in Δ are typed at time τ and the conclusion is at time τ, it is enough to "move to" time τ and then type w.r.t that timestep. Finally, we have (I_τ-E) and ($\otimes_\tau$-E). These allow us to work with linear unit and products at time τ. These are added explicitly since they can not be derived by the other rules, and are needed for typing certain kinds of programs, e.g., see the typing on *turnRedOnClick* below.

Linear Judgments:

$$\frac{}{\Theta; \Gamma; a : A \vdash_l a : A}\ (\textsc{Var}) \qquad \frac{\Theta; \Gamma; \Delta, a : A \vdash_l t : B}{\Theta; \Gamma; \Delta \vdash_l \lambda a.t : A \multimap B}\ (\multimap\text{-I})$$

$$\frac{\Theta; \Gamma; \Delta_1 \vdash_l t_1 : A \multimap B \qquad \Theta; \Gamma; \Delta_2 \vdash_l t_2 : A}{\Theta; \Gamma; \Delta_1, \Delta_2 \vdash_l t_1 t_2 : B}\ (\multimap\text{-E}) \qquad \frac{}{\Theta; \Gamma; \cdot \vdash_l \langle\rangle : \mathsf{I}}\ (\mathsf{I}\text{-I})$$

$$\frac{\Theta; \Gamma; \Delta_1 \vdash_l t_1 : \mathsf{I} \qquad \Theta; \Gamma; \Delta_2 \vdash_l t_2 : C}{\Theta; \Gamma; \Delta_1, \Delta_2 \vdash_l \mathsf{let}\ \langle\rangle = t_1\ \mathsf{in}\ t_2 : C}\ (\mathsf{I}\text{-E}) \qquad \frac{\Theta; \Gamma; \Delta_1 \vdash_l t_1 : A \qquad \Theta; \Gamma; \Delta_2 \vdash_l t_2 : B}{\Theta; \Gamma; \Delta_1, \Delta_2 \vdash_l \langle t_1, t_2 \rangle : A \otimes B}\ (\otimes\text{-I})$$

$$\frac{\Theta; \Gamma; \Delta_1 \vdash_l t_1 : A \otimes B \qquad \Theta; \Gamma; \Delta_2, a : A, b : B \vdash_l t_2 : C}{\Theta; \Gamma; \Delta_1, \Delta_2 \vdash_l \mathsf{let}\ (a, b) = t_1\ \mathsf{in}\ t_2 : C}\ (\otimes\text{-E}) \qquad \frac{\Theta; \Gamma; \Delta \vdash_l t : A}{\Theta; \Gamma; \Delta \vdash_l \mathsf{evt}\ t : \Diamond A}\ (\Diamond\text{-I})$$

$$\frac{\Theta; \Gamma; \Delta \vdash_l t_1 : \Diamond A \qquad \Theta; \Gamma; a : A \vdash_l t_2 : \Diamond B}{\Theta; \Gamma; \Delta \vdash_l \mathsf{let\ evt}\ a = t_1\ \mathsf{in}\ t_2 : \Diamond B}\ (\Diamond\text{-E}) \qquad \frac{\Theta \vdash_i \tau : \mathsf{Time} \qquad \Theta; \Gamma; \Delta \vdash_l t :_\tau A}{\Theta; \Gamma; \Delta \vdash_l t \mathbin{@} \tau : A \mathbin{@} \tau}\ (@\text{-I})$$

$$\frac{\Theta \vdash_i t : \mathsf{Time} \qquad \Theta; \Gamma; \Delta_1 \vdash_l t_1 : A \mathbin{@} \tau \qquad \Theta; \Gamma; \Delta_2, a :_\tau A \vdash_l t_2 : B}{\Theta; \Gamma; \Delta_1, \Delta_2 \vdash_l \mathsf{let}\ a \mathbin{@} \tau = t_1\ \mathsf{in}\ t_2 : B}\ (@\text{-E})$$

$$\frac{\Theta; \Gamma \vdash_c e : \mathsf{G}\ A}{\Theta; \Gamma; \cdot \vdash_l \mathsf{runG}\ e : A}\ (\mathsf{G}\text{-E}) \qquad \frac{\Theta; \Gamma \vdash_c e : X}{\Theta; \Gamma; \cdot \vdash_l \mathsf{F}\ e : \mathsf{F}\ x}\ (\mathsf{F}\text{-I})$$

$$\frac{\Theta; \Gamma; \Delta_1 \vdash_l t_1 : \mathsf{F}\ X \qquad \Theta; \Gamma, x : X; \Delta_2 \vdash_l t_2 : B}{\Theta; \Gamma; \Delta_1, \Delta_2 \vdash_l \mathsf{let\ F}\ x = t_1\ \mathsf{in}\ t_2 : B}\ (\mathsf{F}\text{-E}) \qquad \frac{\Theta, i : \sigma; \Gamma; \Delta \vdash_l t : A}{\Theta; \Gamma; \Delta \vdash_l \Lambda(i : \sigma).t : \forall(i : \sigma).A}\ (\forall\text{-I})$$

$$\frac{\Theta \vdash_i s : \sigma \qquad \Theta; \Gamma; \Delta \vdash_l t : \forall(i : \sigma).A}{\Theta; \Gamma; \Delta \vdash_l t_s : \{s/i\}A}\ (\forall\text{-I}) \qquad \frac{\Theta \vdash_i s : \sigma \qquad \Theta; \Gamma; \Delta \vdash_l t : \{s/i\}A}{\Theta; \Gamma; \Delta \vdash_l \{s, t\} : \exists(i : \sigma).A}\ (\exists\text{-I})$$

$$\frac{\Theta; \Gamma; \Delta_1 \vdash_l t_1 : \exists(i : \sigma).A \qquad \Theta, s : \sigma; \Gamma; \Delta_2, a : \{s/i\}A \vdash_l t_2 : B}{\Theta; \Gamma; \Delta_1, \Delta_2 \vdash_l \mathsf{let\ unpack}\ \{s, a\} = t_1\ \mathsf{in}\ t_2 : B}\ (\exists\text{-E})$$

$$\frac{\begin{array}{c}\Theta; \Gamma; \Delta_1 \vdash_l t_1 : \Diamond A \\ \Theta; \Gamma; \Delta_2 \vdash_l t_2 : \Diamond B \qquad \Theta; \Gamma; a : A, t_2 : \Diamond B \vdash_l t_1' : \Diamond C \qquad \Theta; \Gamma; b : B, t_1 : \Diamond A \vdash_l t_2' : \Diamond C\end{array}}{\Theta; \Gamma; \Delta_1, \Delta_2 \vdash_l \mathsf{select}\ (t_1\ \mathsf{as}\ a \mapsto t_1' \mid t_2\ \mathsf{as}\ b \mapsto t_2') : \Diamond C}\ (\textsc{select})$$

$$\frac{\Theta \vdash_i \tau : \mathsf{Time} \qquad \Delta' = \Delta \downarrow^\tau \qquad \Theta; \Gamma; \Delta' \vdash_l t : A}{\Theta; \Gamma; \Delta \vdash_l t :_\tau A}\ (\textsc{delay})$$

$$\frac{\Theta \vdash_i \tau : \mathsf{Time} \qquad \Theta; \Gamma; \Delta_1 \vdash_l t_1 :_\tau \mathsf{I} \qquad \Theta; \Gamma; \Delta_2 \vdash_l t_2 : B}{\Theta; \Gamma; \Delta_1, \Delta_2 \vdash_l \mathsf{let}\ \langle\rangle \mathbin{@} \tau = t_1\ \mathsf{in}\ t_2 : B}\ (\mathsf{I}_\tau\text{-E})$$

$$\frac{\Theta \vdash_i \tau : \mathsf{Time} \qquad \Theta; \Gamma; \Delta_1 \vdash_l t_1 :_\tau A \otimes B \qquad \Theta; \Gamma; \Delta_2, a :_\tau A, b :_\tau B \vdash_l t_2 : C}{\Theta; \Gamma; \Delta_1, \Delta_2 \vdash_l \mathsf{let}\ (a, b) \mathbin{@} \tau = t_1\ \mathsf{in}\ t_2 : C}\ (\otimes_\tau\text{-E})$$

Fig. 4. Linear Typing rules

3.2 Unfolding Events to Exists

The type system as given above contains both $\Diamond A$ and $A \mathbin{@} k$, as two different way to handle time. The former denotes that something of type A will arrive at *some* point in the future, whereas the latter denotes that something of type A arrives at a *specific* point in the future. The strength of $\Diamond$ is that is gives easy and concise typing rules, whereas the strength of $A \mathbin{@} k$ is that it allows for a more precise usage of time. To connect these two, we add the linear isomorphism $\Diamond A \cong \exists k. A \mathbin{@} k$ to our language, which is witnessed by out and into, as part of the widget API. This isomorphism is true semantically, but can not be derived in the type system. In particular, this isomorphism allows the select rule to be given with $\Diamond$, while still allowing the use timesteps when working with the resulting event. If we were to give the equivalent definition using timesteps, one would need to have some sort of *constraint system* for deciding which events happens first. Avoiding such a constraint systems also allows for a much simpler implementation, as everything is our type system can be inferred.

3.3 Meta-theory of Substitution

The meta-theory of $\lambda_{\mathsf{Widget}}$ is given in the form of a series of substitution lemmas. Since we have three different contexts, we will end up with six different substitutions into terms. The Cartesian to Cartesian, Cartesian to linear and linear to linear are the usual notion of mutual recursive substitution. More interesting is the substitution of indices into Cartesian and linear terms and types. We prove the following lemma, showing that typing is preserved under index substitution:

Lemma 1 (Preservation of Typing under Index Substitution).

$$\frac{\zeta : \Theta' \to \Theta \qquad \Theta; \Gamma \vdash_c e : X}{\Theta'; \zeta(\Gamma) \vdash_c \zeta(e) : \zeta(X)} \qquad\qquad \frac{\zeta : \Theta' \to \Theta \qquad \Theta; \Gamma; \Delta \vdash_l t :_\tau A}{\Theta'; \zeta(\Gamma); \zeta(\Delta) \vdash_l \zeta(t) :_\tau \zeta(A)}$$

Both are these (and all other cases for substitution) are proved by a lengthy but standard induction over the typing tree. See the technical appendix for full proofs of all six substitution lemmas.

3.4 Typing Example

In the following, we go through the formal typing of the *turnRedOnClick* example from section 2. In the below, we have annotated each line with the contents of the index context (omitting the $i : \mathsf{Id}$ that is given upfront) and the linear context.

```
1  turnRedOnClick : ∀ (i : Id), Widget i ⊸ Widget i
2  turnRedOnClick i w₀ =                                  -- w₀ : Widget i
3    let (w₁, c₀)        = onClick i w₀ in                -- w₁ : Widget i, c₀ : ◇I
4    let unpack (x, c₁)  = out c₀ in                      -- x : Time; w₁ : Widget i, c₁ : I @ x
5    let c₂ @ x          = c₁ in                          -- x : Time; w₁ : Widget i, c₂ :ₓ I
6    let ⟨⟩ @ x           = c₂ in                          -- x : Time; w₁ : Widget i
7    let (p, w₂)         = split i x w₂ in                -- x : Time; p : Prefix i x, w₂ : Widget i @ x
8    let w₃ @ x          = w₂ in                          -- x : Time; p : Prefix i x, w₃ :ₓ Widget i
9    let w₄              = (setColor (F Red) w₃) @ x in   -- x : Time; p : Prefix i x, w₄ : Widget i @ x
10   join i x (p, w₄)
```

Most of the above is simple application of elimination rules. In line 4, we add the indices variable $x : \mathsf{Time}$ to the index context. Note in particular the use of $\mathsf{I}_\tau - E$ in line 6 to discharge $c_2 :_x \mathsf{I}$. The point where we actually modify the widget is in line 9 and 10, where we have the following typing:

$$\dfrac{\dfrac{\Delta_1 \vdash w_3 :_x \mathsf{Widget}\ i}{\Delta_1 \vdash (\mathsf{setColor}\ (\mathsf{F}\ \mathsf{Red})\ w_3) \mathbin{@} x : \mathsf{Widget}\ i \mathbin{@} x} \qquad \dfrac{\Delta_2 \vdash p : \mathsf{Prefix}\ i\ x \qquad \Delta_3 \vdash w_4 : \mathsf{Widget}\ i \mathbin{@} x}{\Delta_2, \Delta_3 \vdash \mathsf{join}\ i\ x\ (p, w_4) : \mathsf{Widget}\ i}}{\Delta_1, \Delta_2 \vdash \mathbf{let}\ w_4\ =\ (\mathsf{setColor}\ (\mathsf{F}\ \mathsf{Red})\ w_3) \mathbin{@} x\ \mathbf{in}\ \mathsf{join}\ i\ x\ (p, w_4) : \mathsf{Widget}\ i}$$

where $\Delta_1 = w_3 :_x \mathsf{Widget}\ i$, $\Delta_2 = p : \mathsf{Prefix}\ i\ x$ and $\Delta_3 = w_4 : \mathsf{Widget}\ i \mathbin{@} x$.

3.5 Logical Interpretation

Our language has a straightforward logical interpretation.

The logic corresponding to the Cartesian fragment is a propositional intuitionistic logic, following the usual Curry–Howard interpretation. The logic corresponding to the substructural part of the language is a linear, linear temporal logic. The single-use condition on variables means that the syntax and typing rules correspond to the rules of intuitionistic linear logic (i.e., the first occurrence of linear in "linear, linear temporal"). However, we do not have a comonadic exponential modality $!A$ as a primitive. Instead, we follow the Benton–Wadler approach [4,3] and decompose the exponential into the composition of a pair of adjoint functors mediating between the Cartesian and linear logic.

In addition to the Benton–Wadler rules, we have a temporal modality $\Diamond A$, which corresponds to the eventually modality of linear temporal logic (i.e., the second occurrence of "linear" in "linear, linear temporal logic"). This connective is usually written $F\,A$ in temporal logic, but that collides with the F modality of the Benton–Wadler calculus. Therefore we write it as $\Diamond A$ to reflect its nature as a possibility modality (or monad). In our calculus, the axioms of S4.3 are derivable:

$$\begin{aligned}(T) &: A \multimap \Diamond A \\ (4) &: \Diamond\Diamond A \multimap \Diamond A \\ (.3) &: \Diamond(A \otimes B) \multimap \Diamond((\Diamond A \otimes B) \oplus \Diamond(A \otimes \Diamond B) \oplus \Diamond(A \otimes B))\end{aligned}$$

Note that because the ambient logic is linear, intuitionistic implication $X \to Y$ is replaced with the linear implication $A \multimap B$, and intuitionistic conjunction $X \wedge Y$ is replaced with the linear tensor product $A \otimes B$. It is easy to see that the first two axiom corresponds to the monadic structure of $\Diamond$, and the .3 axiom corresponds to the select rule (with our syntax for select corresponding to immediately waiting for and then pattern-matching on the sum type). In the literature, the .3 axiom is often written in terms of the box modality $\Box A$ [8], but we present it here in a (classically) equivalent formulation mentioning the eventually modality $\Diamond A$.

We do not need to offer an additional explicit box modality $\Box A$, since the decomposition of the exponential $\mathsf{F}(\mathsf{G}A)$ from the linear-non-linear calculus serves that role.

In our system, *we do not want to offer* the next-step operator $\triangleright A$. Since we want to model asynchronous programming, we do not want to include a facility for permitting programmers to write programs which wake up in a specified amount of time. Instead, we only offer an iterated version of this connective, $A \mathbin{@} n$, which can be interpreted as $\triangleright^n A$, and our term syntax does not have any numeric constants which can be used to demand a specific delay.

Finally, the universal and existential quantifiers (in both the intuitionistic and linear fragments) are the usual quantifier rules for first-order logic.

4 Semantics

In this section we will present a denotational model for $\lambda_{\mathsf{Widget}}$. The model is a linear-non-linear (LNL) hyperdoctrine [24,16] with the non-linear part being Set and the linear part being the category of internal relations over a suitable "reactive" category. The hyperdoctrine structure itself is used to interpret the quantification over indices. It many ways this model is entirely standard, and the most interesting thing is the reactive base category and the interpretation of widgets. It is well known that any symmetric monoidal closed category (SMCC) models multiplicative intuitionistic linear logic (MILL), and it is similarly well known that the category of relations over Set can be give the structure of a SMCC by using the Cartesian product as both the monoidal product and monoidal exponential. This construction lift directly to any category of internal relations over a category that is suitably "Set-like", i.e., a topos. Our base category is a simple presheaf category, and hence, we use this construction to model the linear fragment of $\lambda_{\mathsf{Widget}}$.

4.1 The Base Reactive Category

The base reactive category is where the notion of time will arise and is it this notion that will be lifted all the way up to the LNL hyperdoctrine. The simplest model of "time" is $\mathsf{Set}^{\mathbb{N}}$, which can be understood as "sets through time" [23]. This can indeed by used as a model for a reactive setting, but for our purposes it is too simple, and further, depending on which ordering is considered for $\mathbb{N}$, may have undesirable properties for the reactive setting. Instead, we use the only slightly more complicated $\mathsf{Set}^{\mathbb{N}+1}$, henceforth denoted $\mathcal{R}$, where the ordering on $\mathbb{N}+1$ is the discrete ordering on $\mathbb{N}$ and 1 is related to everything else. Adding this "point at infinity" allows global reasoning about objects, an intuition that is further supported by the definition of the sub-object classifier below. Further, this model is known to be able to differentiate between least and greatest fixpoints [15], and even though we do not use this for $\lambda_{\mathsf{Widget}}$, we consider it a useful property for further work (see section 5). Objects in $\mathcal{R}$ can be visualized as

$$A = \quad \begin{array}{ccccc} & & A_\infty & & \\ & \swarrow^{\pi_1} & \downarrow{\scriptstyle \pi_2} & \searrow & \\ A_0 & & A_1 & & \cdots \end{array}$$

We can think of A_∞ as the global view of the object and A_n as the local view of the object at each timestep. Morphisms are natural transformations between such diagrams and the naturality condition means that having a map from A_∞ to B_∞ must also come with coherent maps at each timestep.

In $\mathcal{R}$ we define two endofunctors, which can be seen as describing the passage of time:

Definition 1. *We define the* later *and* previous *endofunctors on $\mathcal{R}$, denoted $\triangleright$ and $\triangleleft$, respectively:*

$$(\triangleright A)_n := \begin{cases} 1 & n = 0 \\ A_{n'} & n = n' + 1 \\ A_\infty & n = \infty \end{cases} \qquad (\triangleleft A)_n := \begin{cases} A_{n+1} & n \neq \infty \\ A_\infty & n = \infty \end{cases}$$

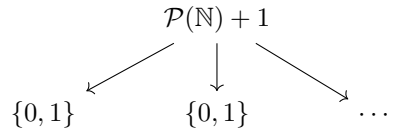
Note that when we apply the later functor, the global view does not change, but the local views are shifted forward in time.

Theorem 1. *The later and previous endofunctors form an adjunction:*

$$\triangleleft \vdash \triangleright$$

Proof. The proof follows easily from an examination of the appropriate diagrams.

Definition 2. *The sub-object classifier, denoted Ω, in $\mathcal{R}$ is the object*



For each $n \in \mathbb{N}$, Ω_n denotes whether a given proposition is true at the nth timestep. Ω_∞ gives the "global truth" of a given proposition. The left injection is some subset of $\mathbb{N}$ that denotes at which points in time something is true. The right injection denotes that something is true "at the limit", and in particular, also at all timesteps. Note, a proposition can be true at all timesteps but not at the limit. This extra point at infinity is precisely what allows us the differentiate between least and greatest fixpoints.

4.2 The Category of Internal Relations

To interpret the linear fragment of the language, we will use the category of internal relations on $\mathcal{R}$. Given two objects A and B in $\mathcal{R}$, an *internal relation* is a sub-object of the product $A \times B$. This can equivalently by understood as a map $A \times B \to \Omega$. The category of internal relations in the category where the objects are the objects of $\mathcal{R}$ and the morphisms $A \to B$ are internal relations $A \times B \to \Omega$ in $\mathcal{R}$. We denote the category of internal relations as $\mathsf{Rel}_\mathcal{R}$.

Definition 3. *We define a monoidal product and monoidal exponential on $\mathsf{Rel}_\mathcal{R}$ as*

$$A \otimes B = A \times B \qquad\qquad A \multimap B = A \times B$$

Theorem 2. *Using the above definition of monoidal product and exponential, $\mathsf{Rel}_\mathcal{R}$ is a symmetric monoidal closed category.*

Proof. All of the properties of the monoidal product and exponential follows easily. Consider the evaluation map $(A \multimap B) \otimes A \to_{\mathsf{Rel}_\mathcal{R}} B$. By definition this is a relation $(A \times B) \times A \sim_\mathcal{R} B$, which is a map $((A \times B) \times A) \times B \to_\mathcal{R} \Omega$. We define this map to be "true" for tuples $(((a, b), a'), b')$ with $a =_\mathcal{R} a' \wedge b =_\mathcal{R} b'$.

Theorem 3. *There is an adjunction in $\mathsf{Rel}_\mathcal{R}$:*

$$\triangleleft \vdash \triangleright$$

where $\triangleleft$ and $\triangleright$ are the lifting of the previous and later functors from $\mathcal{R}$ to $\mathsf{Rel}_\mathcal{R}$.

Definition 4. *We define the* iterated later modality *or the "at" connective as a successive application of the later modality.*

$$\triangleright^0 A = A$$
$$\triangleright^{(k+1)} A = \triangleright(\triangleright^k A)$$

and we will alternatively write $A \mathbin{@} k$ to mean $\triangleright^k A$.

Definition 5. *We define the* event *functor on* $\mathsf{Rel}_{\mathcal{R}}$ *as an application of the iterated later modality.*

$$\begin{aligned} \Diamond A : \mathsf{Rel}_{\mathcal{R}} &\to \mathsf{Rel}_{\mathcal{R}} \\ (\Diamond A)_\infty &= A_\infty \\ (\Diamond A)_n &= \Sigma(k : \mathbb{N}).(\triangleright^k A)_n \end{aligned}$$

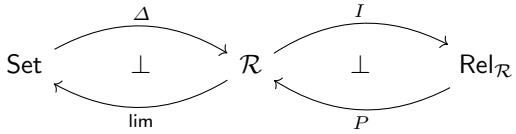
The event functor additionally carries a monadic structure (see [29] and the technical appendix).

Theorem 4. *We have the following isomorphism for any* A

$$\Diamond A \cong \Sigma(n : \mathbb{N}).A \mathbin{@} n$$

Proof. This follows immediately by the two preceding definitions.

Theorem 5. *We have the following adjunctions between* Set, $\mathcal{R}$ *and* $\mathsf{Rel}_{\mathcal{R}}$*:*



where Δ *is the constant functor,* lim *is the limit functor,* I *is the inclusion functor and* P *is the image functor.*

Corollary 1. *The above adjunction induces an adjunction between* Set *and* $\mathsf{Rel}_{\mathcal{R}}$*.*

4.3 The Widget Object

One of the most important objects in $\mathsf{Rel}_{\mathcal{R}}$ is the *widget* object. This object will be used to interpret widgets and prefixes. The widget object will be defined with respect to an ambient notion of identifiers, which we will denote Id. These will be part of the hyperdoctrine structure define below, and for now, we will just assume such an object to exists. We will also use a notion of timesteps internal to the widget object. Note that this timestep is different from the abstract timestep used for defining $\mathsf{Rel}_{\mathcal{R}}$, but are related as defined below. We denote the abstract timesteps with Time.

Before we can define the widget object itself, we need to define an appropriate object of commands. In our minimal Widget API, the only *semantic* commands will be $\mathsf{setColor}, \mathsf{onClick}$ and $\mathsf{onKeypress}$. The rest of the API will be defined as morphisms on the widget object itself. To work with the semantics commands, we additionally need a *compatibility* relation. This relation describes what commands can be applied at the same time. In our setting this relation is minimal, but can in principle be used to encode whatever restrictions is needed for a given API.

Definition 6. *We define the command object as*

$$\mathsf{Cmd} = \{(\mathsf{setColor}, \mathit{color}), \mathsf{onClick}, \mathsf{onKeypress}\}$$

where color is some element of an "color" object. We define the compatibility relations as

$$\mathit{cmd} \bowtie \mathit{cmd}' \text{ iff } \mathit{cmd} = (\mathsf{setColor}, c) \Rightarrow \mathit{cmd}' \neq (\mathsf{setColor}, c')$$

The only non-compatible combination of commands is two application of the setColor command, the idea being that you can not set the color twice in the same timestep.

We can now define the widget and prefix objects

Definition 7. *The widget object, denoted* Widget*, is indexed by* $i \in \mathsf{Id}$ *and is defined as*

$$\begin{aligned}\mathsf{Widget}_\infty\ i &= \{(w,i) \mid w \in \mathcal{P}(\mathsf{Time} \times \mathsf{Cmd}), (t,c) \in w \wedge (t,c') \in w \to c \bowtie c'\}\\ \mathsf{Widget}_n\ i &= \{(w,i) \subset \mathsf{Widget}_\infty\ i \mid \forall (t,c) \in w, t \leqslant n\}\end{aligned}$$

The prefix object, denoted Prefix*, is indexed by* $i \in \mathsf{Id}$ *and* $t \in \mathsf{Time}$ *and is defined as*

$$\begin{aligned}\mathsf{Prefix}_\infty\ i\ t &= \{(P,i) \subset \mathsf{Widget}_\infty\ i \mid \forall (t',c) \in P, t' \leqslant t\}\\ \mathsf{Prefix}_n\ i\ t &= \begin{cases}\{(P,i) \subset \mathsf{Prefix}_\infty\ i\ t \mid \forall (t',c) \in P, t' \leqslant n\} & n < t\\ \mathsf{I} & \textit{otherwise}\end{cases}\end{aligned}$$

The widget object is basically a collection of times and commands keeping track of what has happened to the widget at various times. One can think of an *logbook* with entries for each time step. At the point at infinity, the "global" behavior of the widget is defined, i.e., the full logbook of the widget. For each n, Widget_n is simply what has happened to the widget so far, i.e., a truncated logbook. The prefix object is a widget object that is only defined up to some timestep, and is the unit after that.

Observe there is a semantic difference between the widget where the color is set only once, and the widget where the color is set at every timestep, and this reflects a real difference in actual widget behavior. The difference between *turnRedOnClick* w and *keepTurningRed* w is that if the former is later set to be blue, it will remain blue, whereas the latter will turn back to being red.

To work with widgets we define two "restriction" maps, which are used later for the interpretations.

Definition 8. *We define the following on widgets and prefixes*

$$\mathsf{shift}\ t : \mathsf{Widget}\ i \to_{\mathsf{Rel}_\mathcal{R}} \mathsf{Widget}\ i \qquad \mathsf{prefix}\ t\ i : \mathsf{Widget}\ i \to_{\mathsf{Rel}_\mathcal{R}} \mathsf{Prefix}\ i\ t$$

$$(\mathsf{shift}\ t\ W)_n = \{(t'-t,c) \mid (t',c) \in W \wedge t \leqslant t'\} \qquad (\mathsf{prefix}\ t\ i\ W)_n = \begin{cases}\{(t',c) \in W \mid t' < t\} & n < t\\ \mathsf{I} & n \geqslant t\end{cases}$$

The intuition behind these is that $\mathsf{prefix}\ t\ i$ "cuts off" the widget after t, giving a prefix, whereas $\mathsf{shift}\ t$ shifts forward all entries in the widget by t.

Using the above, we can now define the split and join morphisms. These are again given w.r.t ambient Id and Time objects, which will be part of the full hyperdoctrine structure:

Definition 9. *We define the following morphisms on the widget object*

$$\mathsf{split}\ i\ t : \mathsf{Widget}\ i \to_{\mathsf{Rel}_\mathcal{R}} \mathsf{Prefix}\ i\ t \otimes \mathsf{Widget}\ i\ @\ t \qquad \mathsf{join}\ i\ t : \mathsf{Prefix}\ i\ t \otimes \mathsf{Widget}\ i\ @\ t \to_{\mathsf{Rel}_\mathcal{R}} \mathsf{Widget}\ i$$

$$(\mathsf{split}\ i\ t\ w)_n = (\mathsf{prefix}\ t\ i\ w, \mathsf{shift}\ t\ w)_n \qquad (\mathsf{join}\ i\ t\ (p,w))_n = \begin{cases}p_n & n < t\\ w_{n-t} & n \geqslant t\end{cases}$$

4.4 Linear-non-linear Hyperdoctrine

So far we have not explained in details how to model the quantifiers in our system. To do this, we use the notion of a *hyperdoctrine* [22]. For ordinary first-order logic, this is a functor from a category of contexts and substitutions to the category of Cartesian closed categories, with the idea being that we have one CCC for each valuation of the free first-order variables.

As our category of contexts, we use a Cartesian category that can interpret our index objects, namely Time and Id, where the former is interpreted as $\mathbb{N}+1$ and the latter as $\mathbb{N}$. In our case, both Set and $\mathsf{Rel}_{\mathcal{R}}$ are themselves hyperdoctrines w.r.t to this category of contexts, the former a first-order hyperdoctrine and the latter a multiplicative intuitionistic linear logic (MILL) hyperdoctrine. Together these form a linear-non-linear hyperdoctrine through the adjunction given in Corollary 1. Formally, we have

Definition 10. *A linear-non-linear hyperdoctrine is a MILL hyperdoctrine L together with a first-order hyperdoctrine C and a fiber-wise monoidal adjunction $F : L \leftrightarrows C : G$.*

Theorem 6. *The categories Set and $\mathsf{Rel}_{\mathcal{R}}$ form a linear-non-linear hyperdoctrine w.r.t the interpretation of the indices objects, with the adjunction given as in Corollary 1.*

We refer the reader to the accompanying technical appendix for the full details.

4.5 Denotational Semantics

We the above, we have enough structure to give an interpretation of $\lambda_{\mathsf{Widget}}$. Again, most of this interpretation is standard in the use of the hyperdoctrine structure, and we interpret $\Diamond$ in the obvious way using the linear hyperdoctrine structure on $\mathsf{Rel}_{\mathcal{R}}$. As an example, we sketch the interpretation of the widget object and the $\mathsf{setColor}$ command below.

Definition 11. *We interpret the* $\mathsf{Widget}\ i$ *and* $\mathsf{Prefix}\ i$ *types using the widget and prefix objects:*

$$[\![\Theta \vdash \mathsf{Widget}\ i]\!] = \mathsf{Widget}\ [\![\Theta \vdash_s i : \mathsf{Id}]\!]$$
$$[\![\Theta \vdash \mathsf{Prefix}\ i\ t]\!] = \mathsf{Prefix}\ [\![\Theta \vdash_s i : \mathsf{Id}]\!]\ [\![\Theta \vdash_s t : \mathsf{Time}]\!]$$

and we interpret the $\mathsf{setColor}$ *commands as:*

$$[\![\mathsf{setColor} : \forall(i : \mathsf{Id}), \mathsf{Widget}\ i \otimes \mathsf{F}\ \mathsf{Color} \multimap \mathsf{Widget}\ i]\!] =$$
$$\{w \cup_W \{(0, (\mathsf{setColor}, col))\} \mid w \in [\![\mathsf{Widget}\ i]\!], col \in [\![\mathsf{Color}]\!]\}$$

where $\cup_W$ is a "widget union", which is a union of sets such that identifiers indices and compatibility of commands are respected

This interpretation shows that a widget is indeed a logbook of events. Using the $\mathsf{setColor}$ command simply adds an entry to the logbook of the widget. Note we only set the color in the current timestep. To set the color in the future, we combine the above with appropriate uses of splits and joins. The interpretation of split and join are done using their semantic counterparts, and the interpretation of $\mathsf{onClick}$ and $\mathsf{onKeypress}$ are done, using our non-deterministic semantics, by associating a widget with *all possible occurrences* of the corresponds event.

4.6 Soundness of Substitution

Finally, we prove that semantic substitution is sound w.r.t syntactic substitution. As with the proofs of type preservation for syntactic substitution, there are several cases for the different kinds of substitution, but the main results is again concerned with substitution of indices:

Theorem 7. *Given* $\zeta : \Theta' \to \Theta$, $\Theta; \Gamma \vdash_c e : X$ *and* $\Theta; \Gamma; \Delta \vdash_l t : A$ *then*

$$[\![\zeta]\!]\,[\![\Theta; \Gamma \vdash_c e : X]\!] = [\![\Theta'; \zeta(\Gamma) \vdash_c \zeta(e) : \zeta(X)]\!]$$
$$[\![\zeta]\!]\,[\![\Theta; \Gamma; \Delta \vdash_l t : A]\!] = [\![\Theta'; \zeta(\Gamma); \zeta(\Delta) \vdash_l \zeta(t) : \zeta(A)]\!]$$

Proofs for all six substitutions lemmas can be found in the technical appendix.

5 Related and Future Work

Much work has sought to offer a logical perspective on FRP via the Curry–Howard correspondence [21,18,17,19,20,10,1]. As mentioned earlier, most of this work has focused on calculi that have a Nakano-style later modality [25], but this has the consequence that it makes it easy to write programs which wake up on every clock tick.

In this paper, we remove the explicit next-step modality from the calculus, which opens the door to a more efficient implementation style based on the so-called "push" (or event/notification-based) implementation style. Elliott [12] also looked at implementing a push-based model, but viewed it as an optimization rather than a first-class feature in its own right. We also hope to use an effect system to track when reflows and redraws occur, which should make it easier to keep track of when potentially expensive UI operations are taking place. Moreover, we can extend the widget semantics and compatibility relation to track these events, which should let us test if we can easily put the domain knowledge of browser developers into our semantic model.

In future work, we plan on implementing a language based upon this calculus, with the idea that we can compile to Javascript, and represent widgets with DOM nodes, and represent the $\Diamond A$ and $A\,@\,n$ temporal connectives using doubly-negated callback types (in Haskell notation, `Event A = (A -> IO ()) -> IO ()`). This should let us write GUI programs in a convenient functional style, while generating code which attaches callbacks and does mutable updates in the same style that a handwritten GUI program would use.

Our model, in terms of $\mathsf{Set}^{\mathbb{N}+1}$, can be seen as a model of LTL that has been enriched from being about time-indexed true-values to time-indexed sets. The addition of the global view or point at infinity has the benefit that it enables us to give a model that distinguishes between least and greatest fixed points [15] (i.e., inductive and coinductive types), unlike in models of guarded recursion where guarded types are bilimit-compact [6]. This will let us use the idea that temporal liveness and safety properties can be encoded in types using inductive and coinductive types [10,2].

One interesting recent development in natural deduction systems for comonadic modalities is the introduction of the so-called "Fitch-style" calculi [7,11], which offer an alternative to the Pfenning–Davies pattern-style elimination [26] for the box comonad. These calculi have seen successful application to reactive programming languages [1], and one interesting question is whether they extend to Benton–Wadler adjoint calculi as well – i.e., can the $\mathsf{F}\ (X)$ modality be equipped with a direct style eliminator?

References

1. Bahr, P., Graulund, C.U., Møgelberg, R.E.: Simply RaTT: a Fitch-style modal calculus for reactive programming without space leaks. Proceedings of the ACM on Programming Languages **3**(ICFP), 1–27 (2019)
2. Bahr, P., Graulund, C.U., Møgelberg, R.: Diamonds are not forever: Liveness in reactive programming with guarded recursion (2020)
3. Benton, N., Wadler, P.: Linear logic, monads and the lambda calculus. Proceedings 11th Annual IEEE Symposium on Logic in Computer Science pp. 420–431 (1996)
4. Benton, P.N.: A mixed linear and non-linear logic: Proofs, terms and models. In: Pacholski, L., Tiuryn, J. (eds.) Computer Science Logic. pp. 121–135. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
5. Berry, G., Cosserat, L.: The ESTEREL synchronous programming language and its mathematical semantics. In: Brookes, S.D., Roscoe, A.W., Winskel, G. (eds.) Seminar on Concurrency. pp. 389–448. Springer Berlin Heidelberg, Berlin, Heidelberg (1985)
6. Birkedal, L., Møgelberg, R.E., Schwinghammer, J., Støvring, K.: First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. In: In Proc. of LICS (2011)
7. Bizjak, A., Grathwohl, H.B., Clouston, R., Møgelberg, R.E., Birkedal, L.: Guarded dependent type theory with coinductive types. In: International Conference on Foundations of Software Science and Computation Structures. pp. 20–35. Springer (2016)
8. Blackburn, P., de Rijke, M., Venema, Y.: Modal Logic. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press (2002)
9. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: LUSTRE: A Declarative Language for Real-time Programming. In: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 178–188. POPL '87, ACM, New York, NY, USA (1987). https://doi.org/10.1145/41625.41641
10. Cave, A., Ferreira, F., Panangaden, P., Pientka, B.: Fair Reactive Programming. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 361–372. POPL '14, ACM, San Diego, California, USA (2014). https://doi.org/10.1145/2535838.2535881
11. Clouston, R.: Fitch-style modal lambda calculi. In: Baier, C., Dal Lago, U. (eds.) Foundations of Software Science and Computation Structures. pp. 258–275. Springer International Publishing, Cham (2018)
12. Elliott, C.: Push-pull functional reactive programming. In: Haskell Symposium (2009), http://conal.net/papers/push-pull-frp
13. Elliott, C., Hudak, P.: Functional reactive animation. In: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming. pp. 263–273. ICFP '97, ACM, New York, NY, USA (1997). https://doi.org/10.1145/258948.258973
14. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1 – 101 (1987). https://doi.org/10.1016/0304-3975(87)90045-4
15. Graulund, C.: Lambda Calculus for Reactive Programming. Master's thesis, IT University of Copenhagen (2018)
16. Haim, M., Malherbe, O.: Linear Hyperdoctrines and Comodules. arXiv e-prints arXiv:1612.06602 (Dec 2016)
17. Jeffrey, A.: LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In: Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012. pp. 49–60. Philadelphia, PA, USA (2012). https://doi.org/10.1145/2103776.2103783
18. Jeltsch, W.: Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. Electronic Notes in Theoretical Computer Science **286**, 229–242 (2012)
19. Jeltsch, W.: Temporal Logic with "Until", Functional Reactive Programming with Processes, and Concrete Process Categories. In: Proceedings of the 7th Workshop on Programming Languages Meets Program Verification. pp. 69–78. PLPV '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2428116.2428128

20. Krishnaswami, N.R.: Higher-order Functional Reactive Programming Without Spacetime Leaks. In: Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming. pp. 221–232. ICFP '13, ACM, Boston, Massachusetts, USA (2013). https://doi.org/10.1145/2500365.2500588
21. Krishnaswami, N.R., Benton, N.: Ultrametric semantics of reactive programs. In: 2011 IEEE 26th Annual Symposium on Logic in Computer Science. pp. 257–266. IEEE Computer Society, Washington, DC, USA (June 2011). https://doi.org/10.1109/LICS.2011.38
22. Lawvere, F.W.: Adjointness in foundations. Dialectica **23**(3-4), 281–296 (1969). https://doi.org/10.1111/j.1746-8361.1969.tb01194.x
23. MacLane, S., Moerdijk, I.: Sheaves in Geometry and Logic: A First Introduction to Topos Theory. Universitext, Springer New York (1994). https://doi.org/10.1007/978-1-4612-0927-0
24. Maietti, M.E., de Paiva, V., Ritter, E.: Categorical models for intuitionistic and linear type theory. In: Tiuryn, J. (ed.) Foundations of Software Science and Computation Structures. pp. 223–237. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
25. Nakano, H.: A modality for recursion. In: Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332). pp. 255–266. IEEE Computer Society, Washington, DC, USA (June 2000). https://doi.org/10.1109/LICS.2000.855774
26. Pfenning, F., Davies, R.: A judgmental reconstruction of modal logic. Mathematical Structures in Computer Science **11**(4), 511–540 (2001). https://doi.org/10.1017/S0960129501003322
27. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science (sfcs 1977). pp. 46–57. IEEE (1977)
28. Pouzet, M.: Lucid Synchrone, version 3. Tech. rep., Laboratoire d'Informatique de Paris (2006)
29. Szamozvancev, D.: Semantics of temporal type systems. Master's thesis, University of Cambridge (2018)